НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

До захисту допущено

Завідувач кафедри

_____ О.В. Коваль
(підпис) (ініціали, прізвище)

“___”_____2020р.

ДИПЛОМНА РОБОТА

на здобуття ступеня бакалавра

з напряму підготовки 122 Комп’ютерні науки та інформаційні технології

на тему Побудова тривимірних моделей з використанням технологій Maya

Виконав: студент 4 курсу, групи ТР-62

Лобус Юрій Юрійович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник д.т.н. Аушева Наталія Миколаївна _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент к.т.н., ст.викл. Рачинський Артур Юрійович _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2020

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти перший рівень

Напрям підготовки 122 Комп’ютерні науки та інформаційні технології

Спеціалізація Геометричне моделювання в інформаційних системах

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Олександр Коваль
(підпис)
”___”_____2020р.

ЗАВДАННЯ

на дипломну роботу студенту

Лобусу Юрію Юрійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Побудова тривимірних моделей з використанням технологій Maya

керівник роботи д.т.н. Аушева Наталія Миколаївна
(прізвище, ім’я, по батькові науковий ступінь, вчене звання)

затверджена наказом вищого навчального закладу від ”__”___ 2020р. № __

2. Строк подання студентом роботи _____

3. Вихідні дані до роботи середовище Autodesk® Maya®, мова програмування Python, Maya Python API, бібліотека polyMidifiers.py

4.Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) аналіз систем 3D графіки, в яких можлива побудова полігональних об’єктів на предмет існуючих рішень, методів та алгоритмів для створення фасок, аналіз можливостей розробки архітектури програмного продукту в середовищі Autodesk® Maya®, розробка доповненого та покращеного алгоритму для створення адаптивних фасок на полігональному об’єкті

5. Перелік ілюстративного матеріалу

Побудова тривимірних моделей з використанням технологій Maya, Вступ, Мета, Аналіз реалізації операції створення фасок в Autodesk® Maya®, Можливості операції для адаптивного створення фасок, Засоби розробки програмного продукту в середовищі Autodesk® Maya®, Архітектура програмного продукту, Опис алгоритму створення адаптивних фасок, Демонстрація роботи плагіна, Висновки

6. Дата видачі завдання ”11” жовтня 2019 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітки
1.	Затвердження теми роботи	01.10.2019 р.	
2.	Вивчення та аналіз задачі	15.11.2019 р.	
3.	Розробка архітектури та загальної структури системи	15.01.2020 р.	
4.	Розробка структур окремих підсистем	15.02.2020 р.	
5.	Програмна реалізація системи	15.03.2020 р.	
6.	Оформлення пояснювальної записки	25.05.2020 р.	
7.	Захист програмного продукту	11.05.2020 р.	
8.	Передзахист	08.06.2020 р.	
9.	Захист	.06.2020 р.	

Студент _____ Лобус Ю. Ю.
(підпис) (прізвище та ініціали,)

Керівник роботи _____ Аушева Н. М.
(підпис) (прізвище та ініціали,)

АНОТАЦІЯ

Операція Bevel є однією з найчастіших у використанні операцій під час моделювання різних поверхонь. Для швидкості та комфорту в роботі з 3D об’єктом операція повинна мати достатньо гнучкі налаштування. Проте в середовищі Autodesk® Maya® методи, що пропонуються за замовчуванням мають недостатню функціональність. Саме тому розробка покращеного алгоритму операції Bevel в середовищі Autodesk® Maya® є актуальною задачею на сьогоднішній день.

Мета роботи – аналіз систем 3D графіки, в яких можлива побудова полігональних об’єктів на предмет існуючих рішень, методів та алгоритмів для створення фасок, аналіз можливостей розробки архітектури програмного продукту в середовищі Autodesk® Maya®, розробка доповненого та покращеного алгоритму для створення адаптивних фасок на полігональному об’єкті.

Записка містить 52 сторінки, 51 рисунок, 3 додатки та 20 посилань.

Ключові слова: 3D ГРАФІКА, AUTODESK MAYA, BEVEL, ФАСКА, ПЛАГІН.

ABSTRACT

The Bevel operation is one of the most common operations in 3D modeling process. For high productivity rate and comfort, such operation should provide user with flexible settings. However default methods in Autodesk® Maya® have lack of functionality. That’s why, nowadays, development of upgraded Bevel algorithm is a relevant goal.

The main purpose of this work is to analyze 3D graphics systems, which provide functional for building polygonal meshes, regarding existing solutions, methods and algorithms of chamfering, analyze software architecture development opportunities in Autodesk® Maya®, developing supplemented and upgraded algorithm of adaptive chamfering on polygonal meshes.

The note contains 52 pages, 51 figures, 3 attachments and 20 links.

Key words: 3D GRAPHICS, AUTODESK MAYA, BEVEL, CHAMFER, PLUG-IN.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

3D – (англ. 3 Dimensions, 3D) – розділ комп'ютерної графіки, сукупність прийомів та інструментів (як програмних, так і апаратних), призначених для зображення об'ємних об'єктів.

Maya® – (Autodesk® Maya®) – графічний редактор, для моделювання тривимірних об'єктів, анімації, композитингу та візуалізації.

API – (англ. Application Programming Interface, API) – набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення.

MEL – (англ. Maya Embedded Language) – це потужна скриптова мова програмування, яка використовується для виклику внутрішніх команд Autodesk® Maya®.

AA – (англ. Anti-Aliasing) – технологія, що використовується в обробці зображень з метою зробити межі кривих ліній більше гладкими, прибираючи «зубці», що виникають на краях об'єктів.

CH – (англ. Construction History) – система історії в Autodesk® Maya®.

DG – (англ. Dependency Graph) це спеціалізована база даних для зберігання графічної інформації.

ВСТУП

Основною задачею 3D графіки із самого її початку була імітація реальних тривимірних поверхонь, матеріалів, світла та їх взаємодії у просторі. З розвитком технологій та ростом обчислювальних потужностей, з’явилась велика кількість програм для створення та обробки 3D графіки. Такі середовища пропонують велику кількість різноманітних інструментів та точних алгоритмів для побудови тривимірних об’єктів.

Проте в природі, на відміну від 3D середовищ, не існує ідеальних математичних форм, як наслідок неможливо зустріти й ідеально гострих кутів. Навіть вироблені людиною на станках деталі мають конструкційні похибки та нерівності. Саме тому важливим інгредієнтом для одержання реалістичного полігонального об’єкту – є фаски.

Окрім косметичної функції фаски можуть також визначати великі форми полігонального об’єкту, що задані кресленням. Також фаски можуть виступати в ролі підтримуючих ребер при роботі зі smooth алгоритмами, наприклад: *Maya Catmull-Clark* або *OpenSubdiv Catmull-Clark* [1].

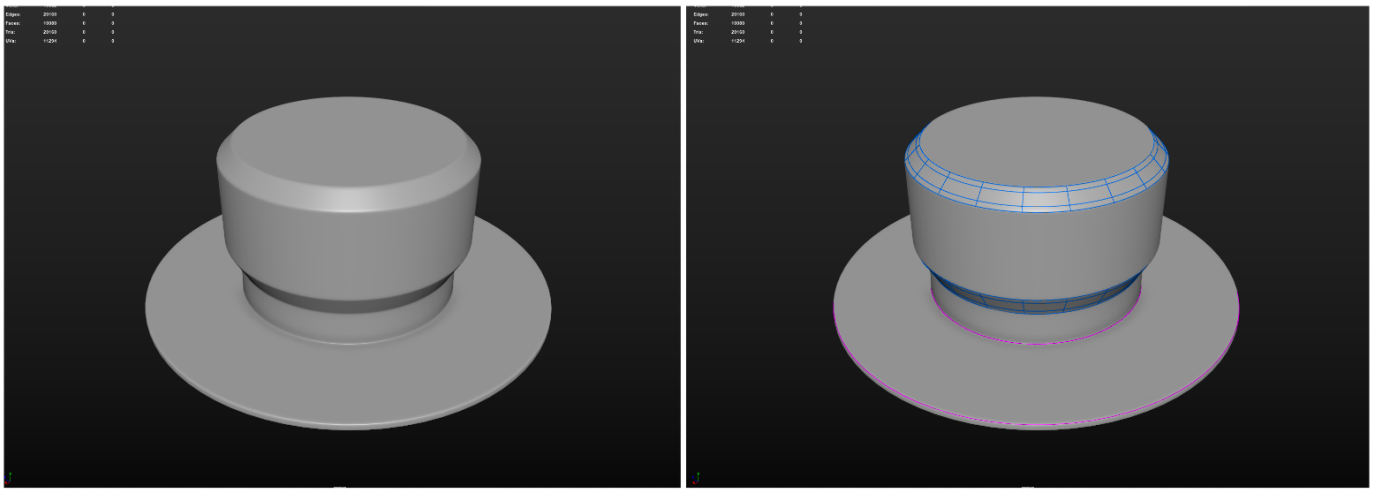

Рис. 1. Приклад різних типів фасок. Синій – фаска, що визначає велику форму. Рожевий – косметична фаска .

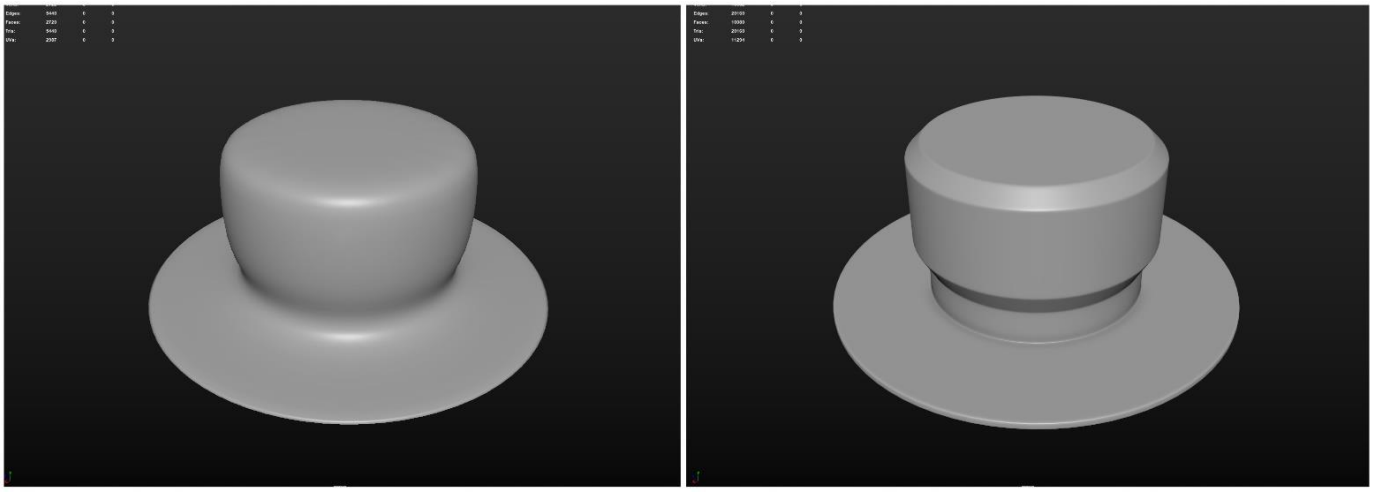

Рис. 2. Демонстрація роботи алгоритму Maya Catmull-Clark. Зліва – без підтримуючих ребер. Справа – з підтримуючими ребрами.

Тому кожний 3D пакет має вбудоване рішення для швидкого та гнучкого створення фасок. Кожна компанія, як правило, має свій особистий алгоритм і деякі з них мають серйозні обмеження. Одним за таких пакетів є Autodesk® Maya®.

Сьогодні, Autodesk® Maya® є індустріальним стандартом для розробки 3D графіки для кіно, телебачення та відеоігор. І використовується такими студіями, як: *Disney Animation, DreamWorks Pictures, Blue Sky Studios* і т. д..

Проте, незважаючи на велику кількість вбудованих рішень та функцій, часто необхідно розширювати можливості пакету, щоб задовільнити специфічні потреби. Оскільки Autodesk® Maya® відкритий до модифікування продукт, кожен незалежний розробник може змінювати або додавати новий функціонал до програмного пакету.

За замовчуванням, Maya® пропонує нам операцію *polyBevel3*[2]. Проблематика існуючого методу полягає в відсутності адаптивності алгоритму операції.

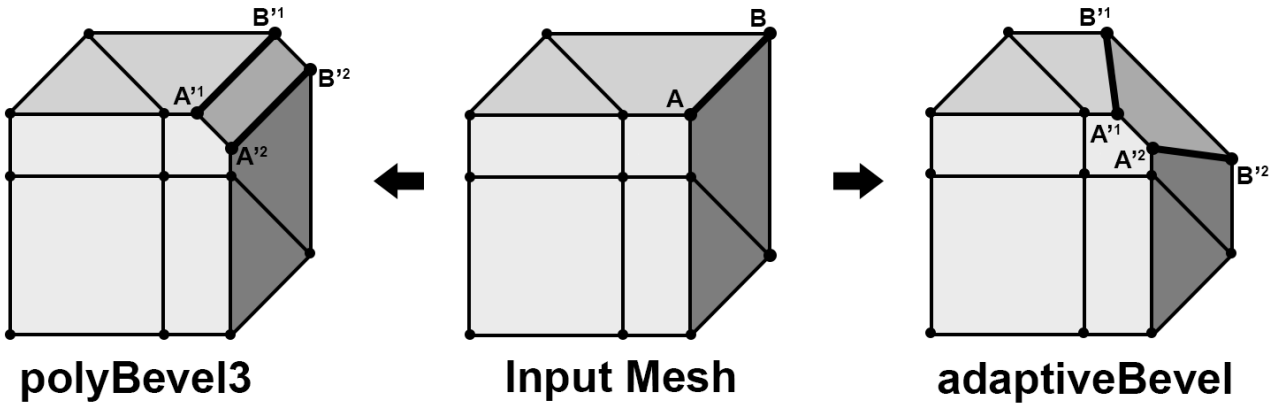

Рис. 3. Порівняння результатів обробки вхідного полігонального об'єкту (Input Mesh).

Таким чином відрізки *A'1A'2* та *B'1B'2* будуть завжди мати однакову довжину незалежно від оточуючої полігональної сітки. А саме, максимальним відхиленням фаски буде відстань до сусідньої точки з найменшою різницею координат. Через це обмеження не завжди можливо відтворити коректну фаску без додаткових маніпуляцій.

Ціль дипломної роботи полягає в аналізі систем 3D графіки, в яких можлива побудова полігональних об'єктів на предмет існуючих рішень, методів та алгоритмів для створення фасок, аналіз можливостей розробки архітектури програмного продукту в середовищі Autodesk® Maya®, розробка доповненого та покращеного алгоритму для створення адаптивних (тобто з урахуванням сусідніх точок) фасок на полігональному об'єкті.

1 ПОСТАНОВКА ЗАДАЧІ СТВОРЕННЯ ФАСОК НА ПОЛІГОНАЛЬНОМУ ОБ'ЄКТІ

1.1 Мета розробки алгоритму адаптивного створення фасок

Мета роботи – аналіз систем 3D графіки, в яких можлива побудова полігональних об'єктів на предмет існуючих рішень, методів та алгоритмів для створення фасок, аналіз можливостей розробки архітектури програмного продукту в середовищі Autodesk® Maya®, розробка доповненого та покращеного алгоритму для створення адаптивних фасок на полігональному об'єкті.

Об'єкт дослідження – обробка та модифікація полігональних об'єктів, методи та алгоритми створення фасок, існуючі рішення операції створення фасок у програмних системах.

Предмет дослідження – середовища розробки Autodesk® Maya®, Autodesk® 3DS Max®, Blender®.

1.2 Загальна структура програмного продукту

Для виконання поставленої задачі, необхідно розробити плагін для адаптивного створення фасок на полігональному об'єкті у середовищі Autodesk® Maya®. Далі такий плагін з покращеним алгоритмом операції *Bevel* буде називатися *adaptiveBevel*.

Для роботи програмного продукту необхідна наступна вхідна інформація:

- існуючий полігональний об'єкт;
- вибрані компоненти існуючого полігонального об'єкту;
- вибраний метод операції;
- вибрані значення атрибутів операції.

Вихідною інформацією вважаються:

- модифікований полігональний об'єкт, з видозміненою сіткою.

Програмний продукт має вирішувати наступні задачі:

- модифікація полігональних об’єктів відповідно до вказаних користувачем атрибутів.
- демонстрація модифікованої сітки у Viewport 2.0[3].
- можливість скасувати роботу операції після її виконання, та повторити після її скасування.

Для роботи над створенням програмного продукту, було вирішено використовувати *Maya Python API*, *Maya Commands* та бібліотеки *polyModifiers.py* на базі мови програмування Python в середовищі розробки PyCharm 2020.

2 АНАЛІЗ РЕАЛІЗАЦІЇ ОПЕРАЦІЇ СТВОРЕННЯ ФАСОК У ІСНУЮЧИХ ПРОГРАМНИХ СИСТЕМАХ

Оскільки створення фасок (Bevel) є поширеною і навіть базовою операцією полігонального моделювання, існує багато прикладів реалізації цієї задачі. В даному розділі описані базові функції операції створення фасок та проаналізовані їх реалізації у існуючих програмних продуктах.

Операція Bevel модифікує ребра так, щоб гострі кути виглядали більш округлими. Алгоритм розраховує нові координати для точок, що приймають участь в операції, та створює грані між новими ребрами. Нормалі точок не змінюються, проте діляться між новими точками (по одній на кожну).

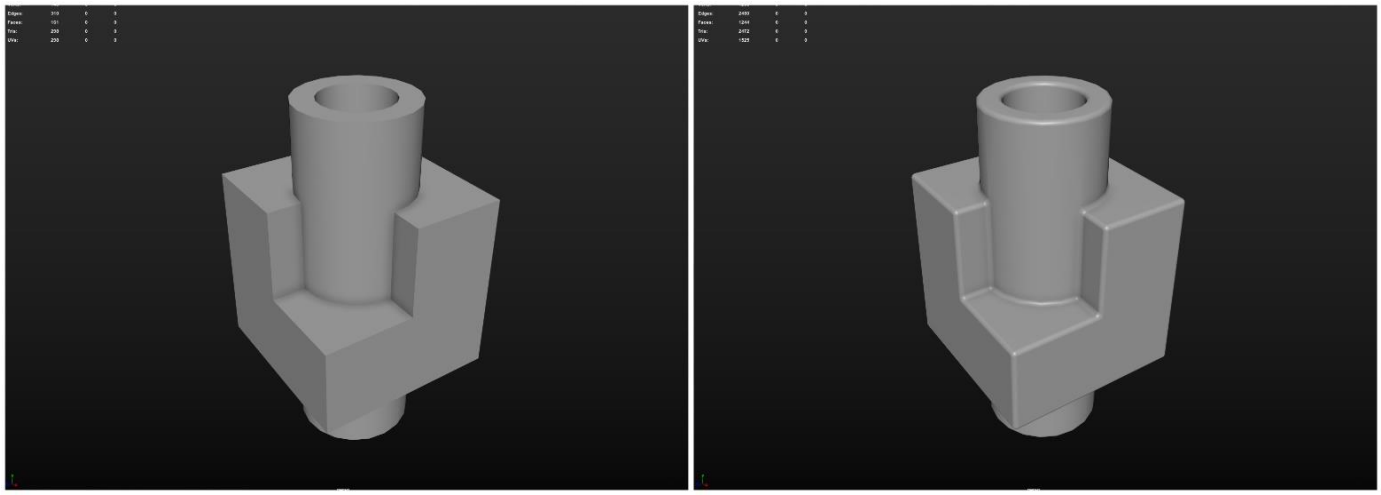

Рис. 4. Демонстрація роботи алгоритму Bevel.

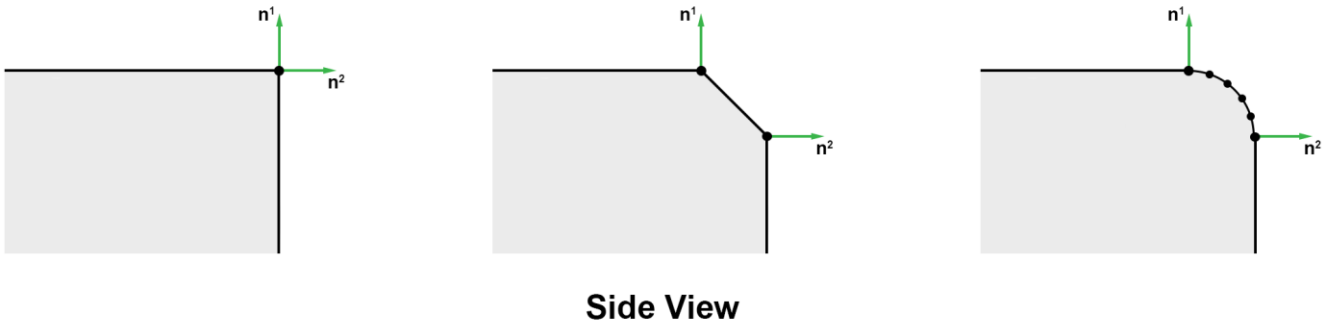

Рис. 5. Нормалі в точках. Зліва – до проведення операції. Справа – після.

2.1 Середовище Autodesk® Maya®

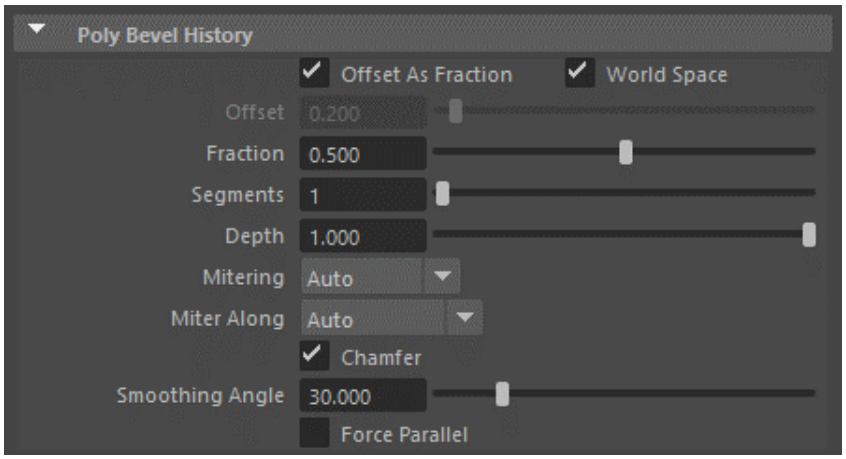

Рис. 6. Вікно налаштувань операції Bevel у середовищі Autodesk® Maya®.

Операція *polyBevel3* в середовищі Autodesk® Maya® має наступний набір основних функцій:

1. *Offset As Fraction* – надає користувачу вибір між двома методами роботи операції за допомогою параметру:
    - *Offset Space* – використовуючи даний метод, розмір фаски на масштабованому об’єкті буде також масштабовано в залежності від величини масштабування полігонального об’єкту.
    - *Fractional* – використовуючи даний метод, розмір фаски буде не більшим за найкоротше прилягаюче ребро. Завдяки цьому методу створюється обмеження на розмір фаски, аби створені грані не проходили наскрізь одне одного.

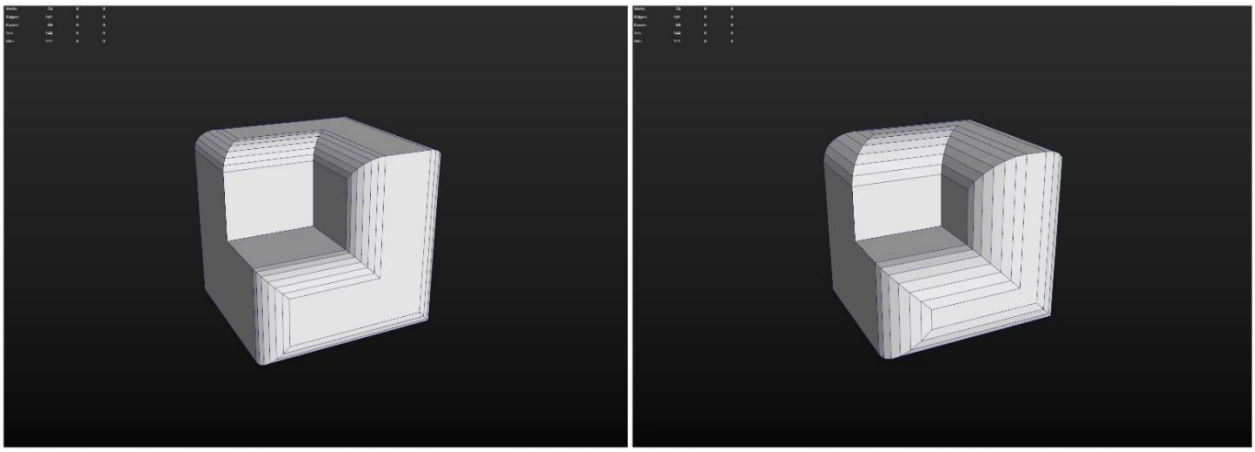

Рис. 7. Наочна різниця між методами у середовищі Autodesk® Maya®. Зліва – з використанням Fraction. Справа – з використанням Offset.

2. *Width* – визначає відстань між оригінальним ребром та центром зміщеної грані, щоб знайти величину фаски. Атрибут *Width* залежить від обраного методу роботи: *Fractional* або *Offset Space*. Коли обраний метод *Fractional*, можливі значення для вводу в поле *Width* обмежені до проміжку від 0 до 1. Коли значення *Width* досягає 1, відстань буде максимально можливою (в залежності від найкоротшого сусіднього ребра), що забезпечує гарантію від несподіваних результатів. Використовуючи метод *Offset Space*, можливо введення будь-яких значень в поле *Width*. Проте робота алгоритму не змінюється а лише знімається обмеження вводу.

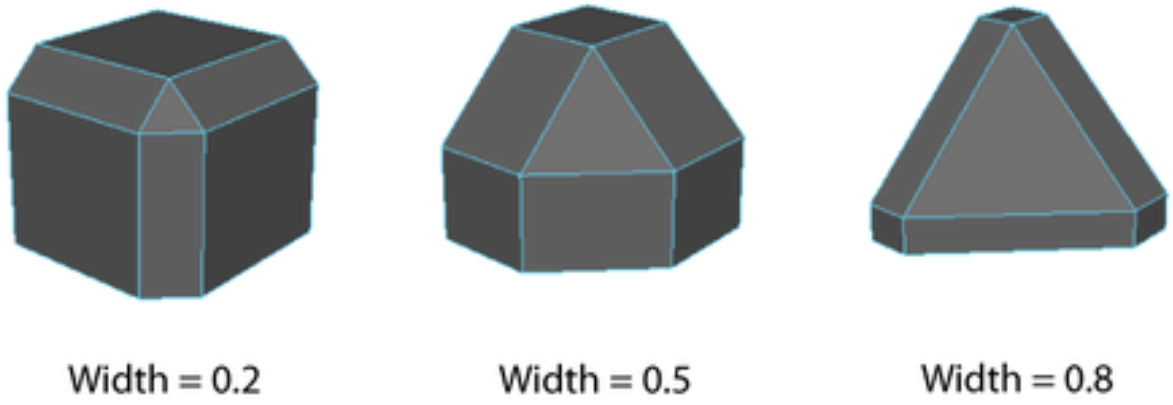

Рис. 8. Демонстрація роботи атрибута width у середовищі Autodesk® Maya® (джерело: http://knowledge.autodesk.com).

3. *Segments* – значення даного атрибуту визначає кількість паралельних поділів нової грані.

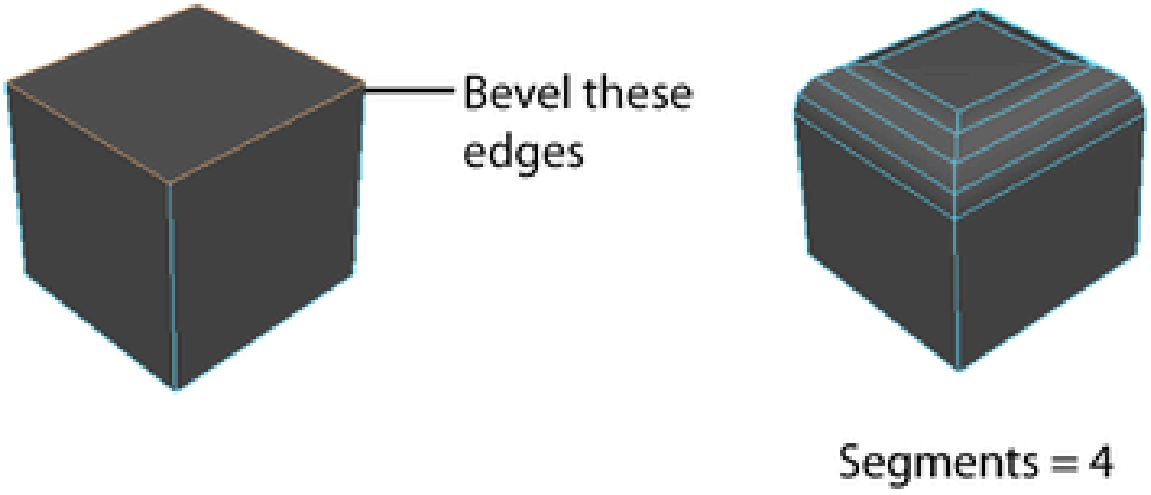

Рис. 9. Демонстрація роботи атрибута segments у середовищі Autodesk® Maya® (джерело: http://knowledge.autodesk.com).

4. *Depth* – визначає напрям вигину (+) або вгину (-) нових граней.

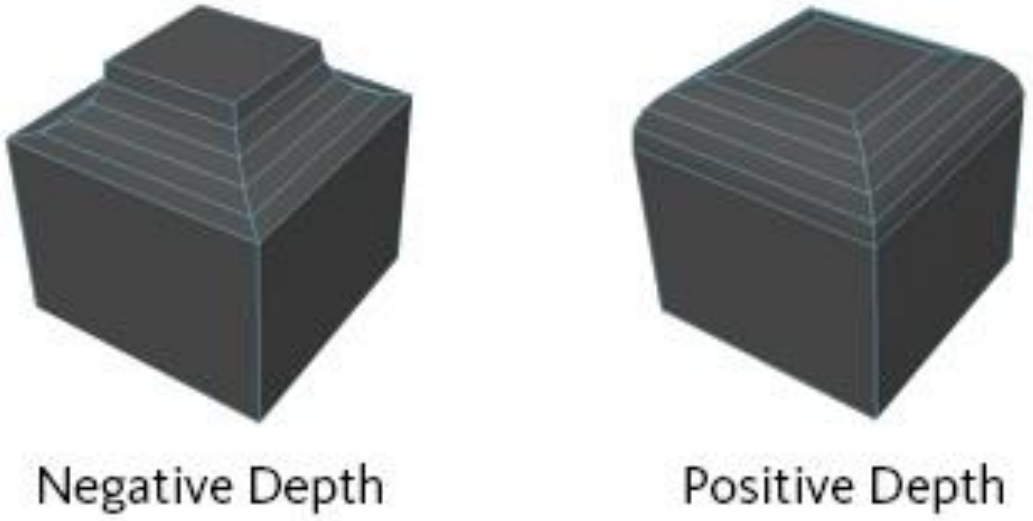

Рис. 10. Демонстрація роботи атрибута depth у середовищі Autodesk® Maya® (джерело: http://knowledge.autodesk.com).

5. *Chamfer* – дозволяє обрати користувачу тип створеної фаски. *Chamfered* додає похилу фаску, а *non-chamfered* – створює підтримуючі грані не змінюючи силуету полігонального об’єкту. Частіше за все *non-chamfered* використовується при подальшій обробці алгоритмами *Maya Catmull-Clark* або *OpenSubdiv Catmull-Clark*.

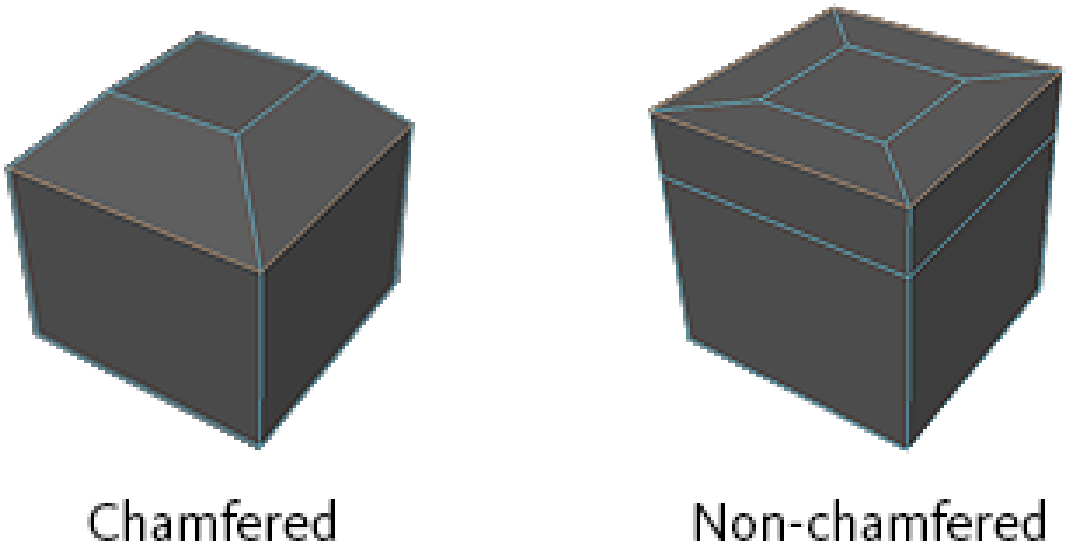

Рис. 11. Демонстрація роботи атрибута chamfer у середовищі Autodesk® Maya® (джерело: http://knowledge.autodesk.com).

На основі аналізу середовища Autodesk® Maya® на предмет створення фасок, можна оцінити проблеми операції *polyBevel3*.

З одного боку метод *Fraction* не допускає непередбачуваних результатів і є універсальним та точним методом створення загальних фасок, проте з іншого боку таке обмеження негативно впливає на гнучкість операції в умовах реальних задач.

Саме тому програмний продукт для адаптивного створення фасок має містити в собі не тільки адаптивний, а й стандартний методи роботи з полігональним об’єктом, задля забезпечення максимального комфорту та гнучкості у використанні на практичних задачах.

2.2 Середовище Autodesk® 3ds Max®

Autodesk® 3DS Max® – графічний 3D редактор, система для створення і редагування 3D об'єктів, розроблена компанією Autodesk®. Проте незважаючи на спільного з Autodesk® Maya® розробника, їх алгоритми створення фасок мають вагомі відмінності.

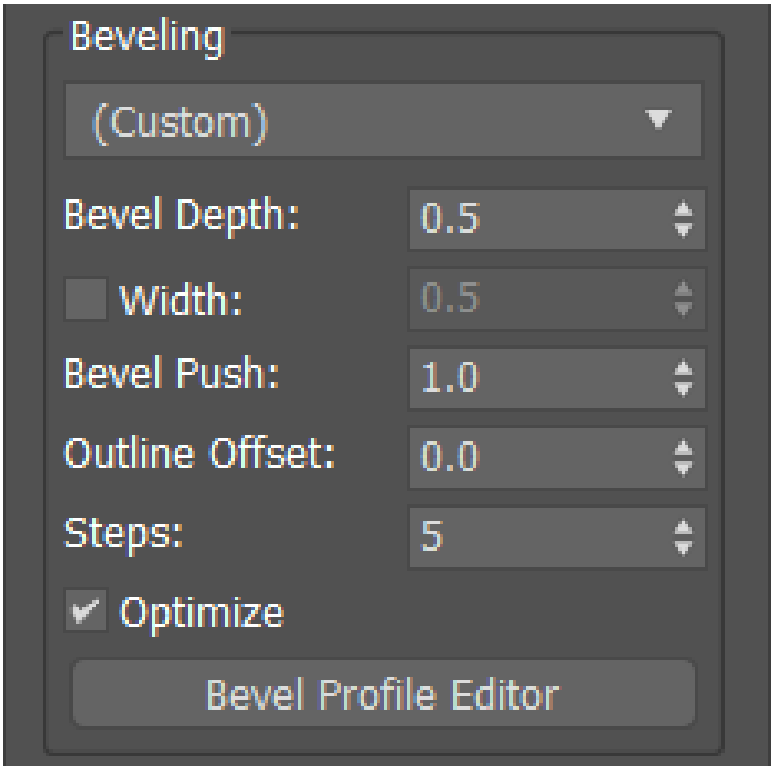

Рис. 12. Вікно налаштувань операції Bevel у середовищі Autodesk® 3DS Max®.

Операція Bevel[4] в середовищі Autodesk® 3ds Max® має наступний набір основних функцій:

1. *Width* – визначає відстань між оригінальним та новим ребром, щоб знайти величину фаски.

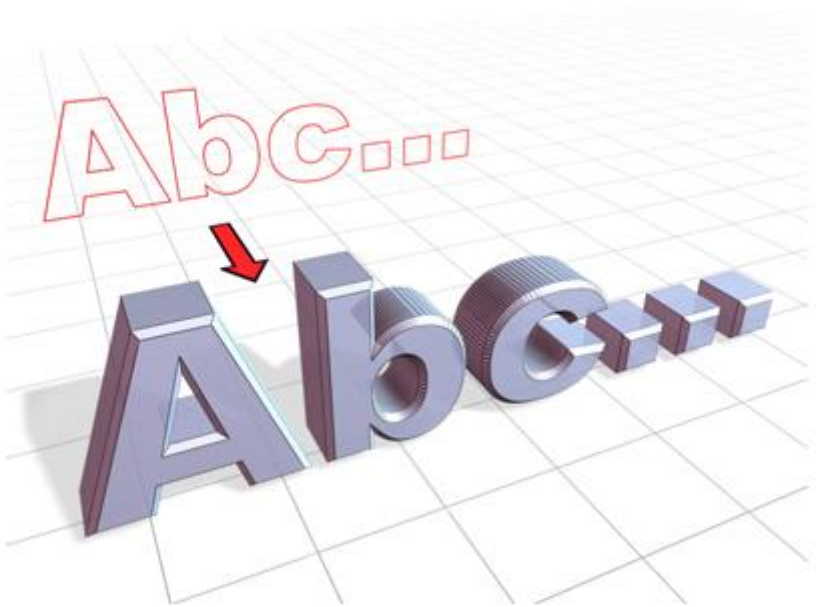

Рис. 13. Демонстрація роботи атрибута width у середовищі Autodesk® 3DS Max® (джерело: http://knowledge.autodesk.com).

2. *Segments* – значення даного атрибуту визначає кількість паралельних поділів нової грані, та має два методи роботи: *Linear* та *Curve*.

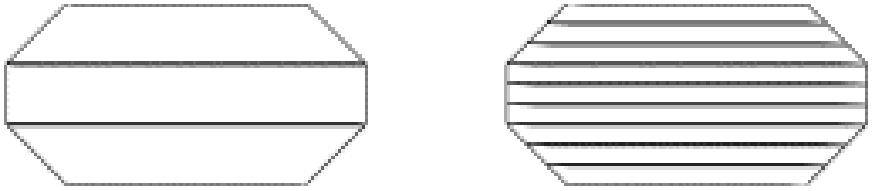

Рис. 14. Демонстрація роботи атрибута segments у середовищі Autodesk® 3DS Max® (джерело: http://knowledge.autodesk.com).

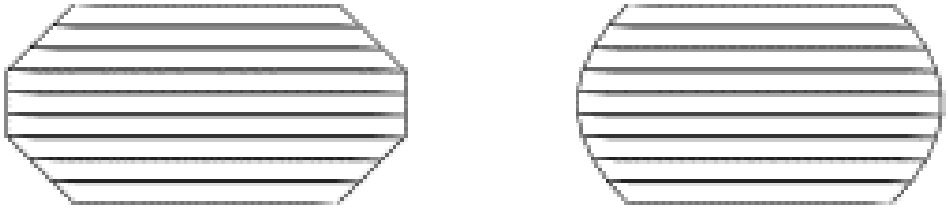

Рис. 15. Демонстрація різниці між методами роботи атрибуту segments у середовищі Autodesk® 3DS Max®. Зліва – linear. Справа – Curve. (джерело: http://knowledge.autodesk.com).

3. *Depth* – визначає напрям вигину (+) або вгину (-) нових граней.

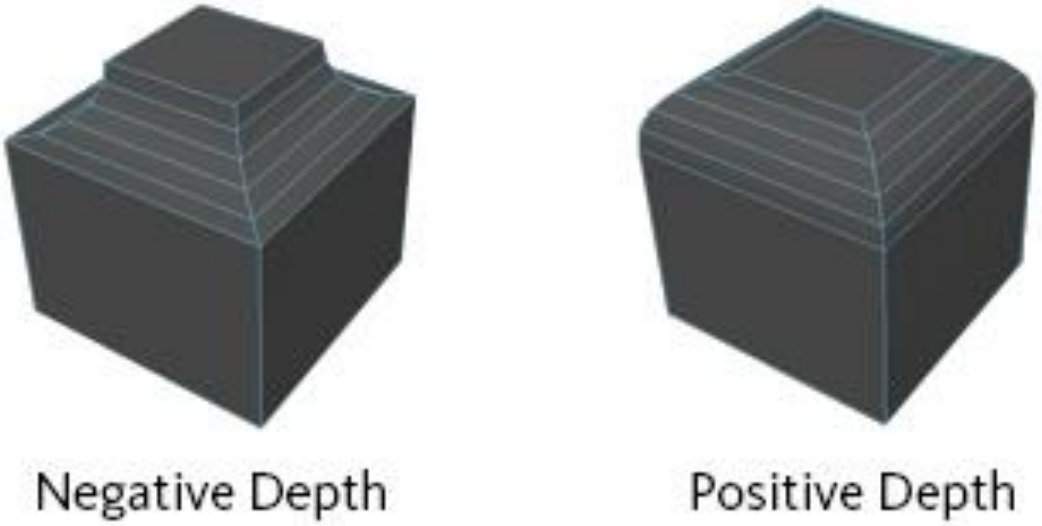

Рис. 16. Демонстрація роботи атрибута depth у середовищі Autodesk® 3DS Max® (джерело: http://knowledge.autodesk.com).

4. *Profile* – видавлює задану іншим об’єктом форму на нових створених гранях.

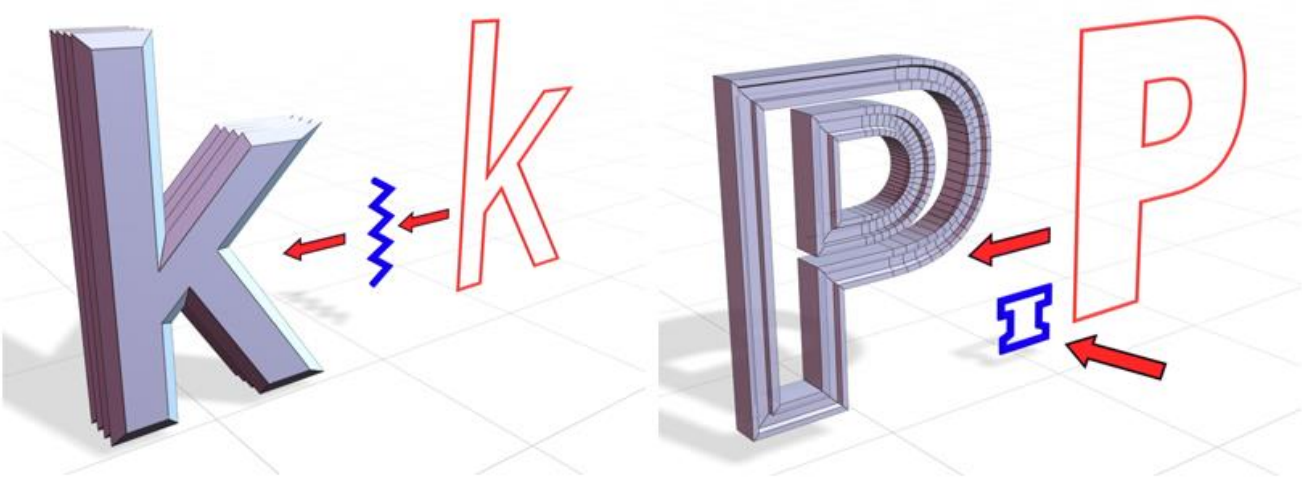

Рис. 17. Демонстрація роботи атрибута profile у середовищі Autodesk® 3DS Max® (джерело: http://knowledge.autodesk.com).

Незважаючи на те, що всі атрибути схожі або повністю повторюють атрибути операції *polyBevel3*, середовища Autodesk® Maya®, алгоритм створення фасок в середовищі Autodesk® 3DS Max® – адаптивний за замовчуванням.

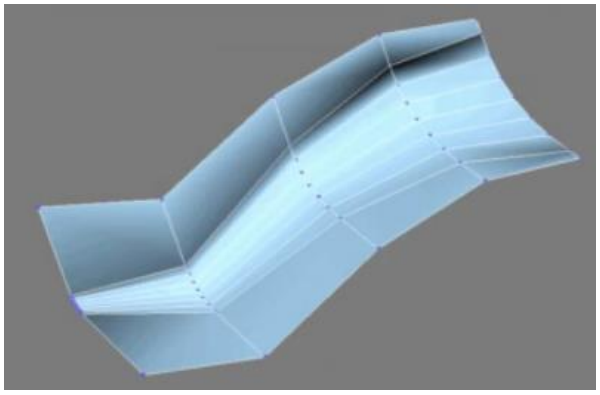

Рис. 18. Демонстрація роботи адаптивного алгоритму операції Bevel у середовищі Autodesk® 3DS Max®.

2.3 Середовище Blender®

Blender® – графічний 3D редактор, система для створення і редагування 3D об'єктів, розроблена компанією Blender Foundation. Операція Bevel[5] в середовищі Blender® має схожий функціонал з Autodesk® Maya® та Autodesk® 3DS Max®, проте якісно виділяється на їх фоні кількістю нових рішень.

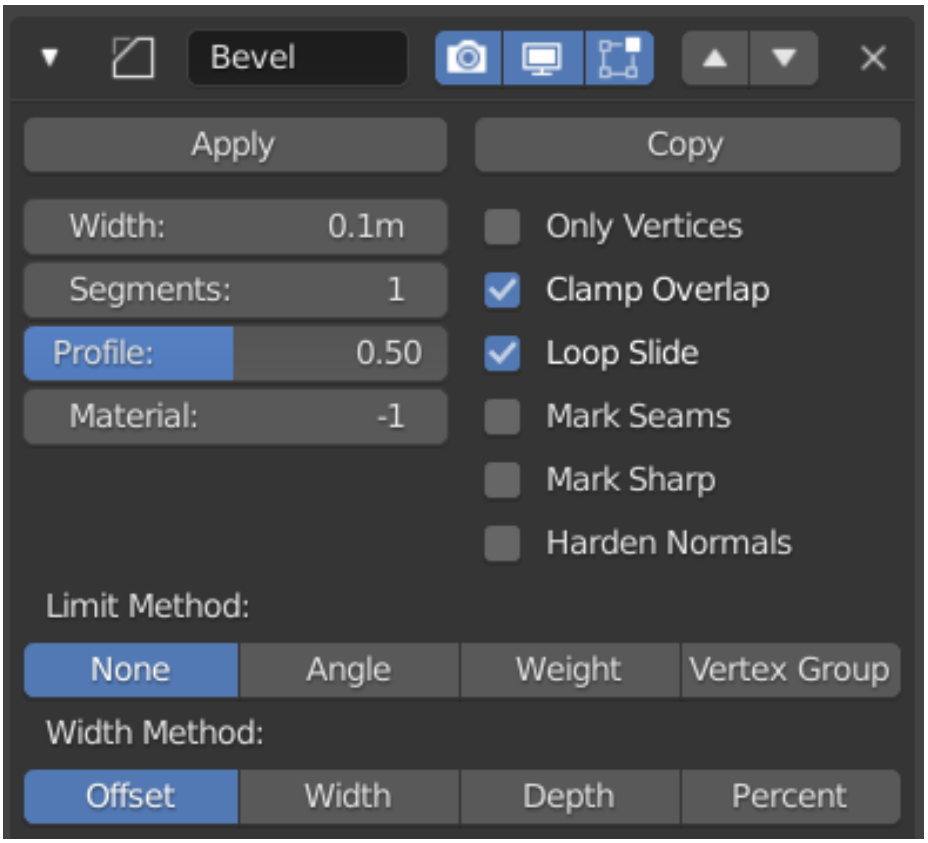

Рис. 19. Вікно налаштувань операції Bevel у середовищі Blender®.

1. Blender® пропонує одразу 4 методи роботи з розміром фаски:
    - Offset – значення розцінюється як відстань від оригінального ребра до ребра грані, що оброблюється операцією Bevel.
    - Width – значення розцінюється як відстань між двома новими ребрами створеними операцією Bevel.
    - Depth - значення розцінюється як перпендикулярна відстань від нової грані, що оброблюється операцією Bevel до оригінального ребра.
    - Percent – аналогічно до методу Offset, проте значення розцінюється як відстань у відсотках до сусіднього ребра.

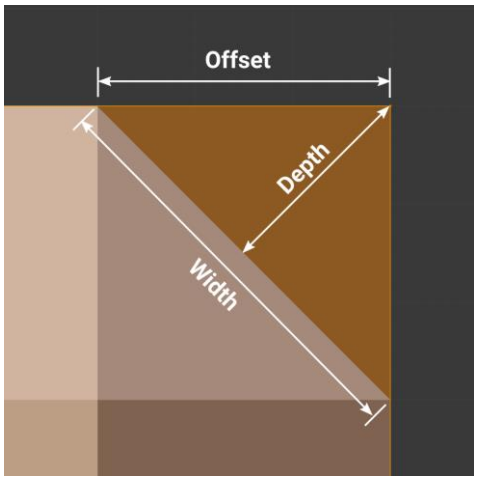

Рис. 20. Схематичне зображення методів роботи операції Bevel в середовищі Blender® (джерело: https://docs.blender.org/).

2. *Width* – визначає відстань між оригінальним та новим ребром, щоб знайти величину фаски.

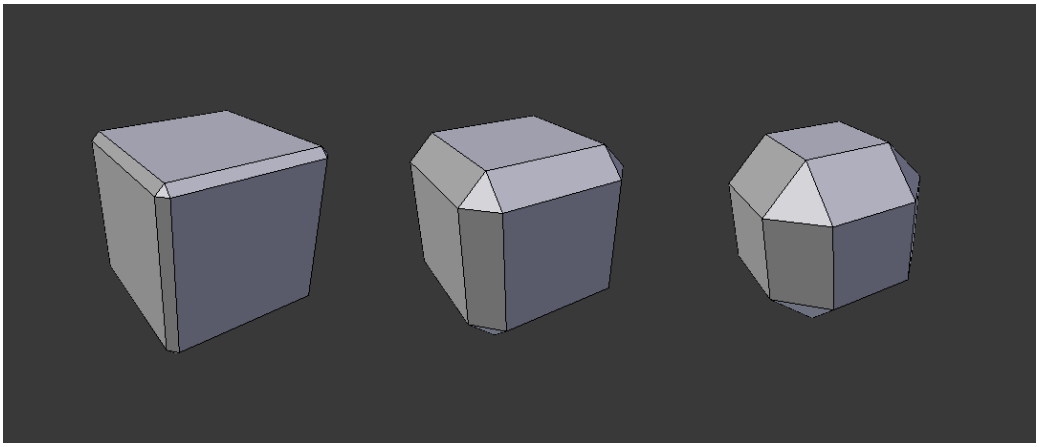

Рис. 21. Демонстрація роботи атрибута width у середовищі Blender® (джерело: https://docs.blender.org/).

3. *Segments* – значення даного атрибуту визначає кількість паралельних поділів нової грані.

4. *Custom Profile* – дозволяє створювати та налаштовувати криві, за якими будуть прямувати нові грані створені операцією *Bevel*. Це надає більшу гнучкість та функціональність ніж атрибут *Depth* в Autodesk® Maya®, і фактично являється його заміною. Змінюючи форму кривої можна швидко створити складну форму, на налаштування якої іншими методами може бути витрачено багато часу.

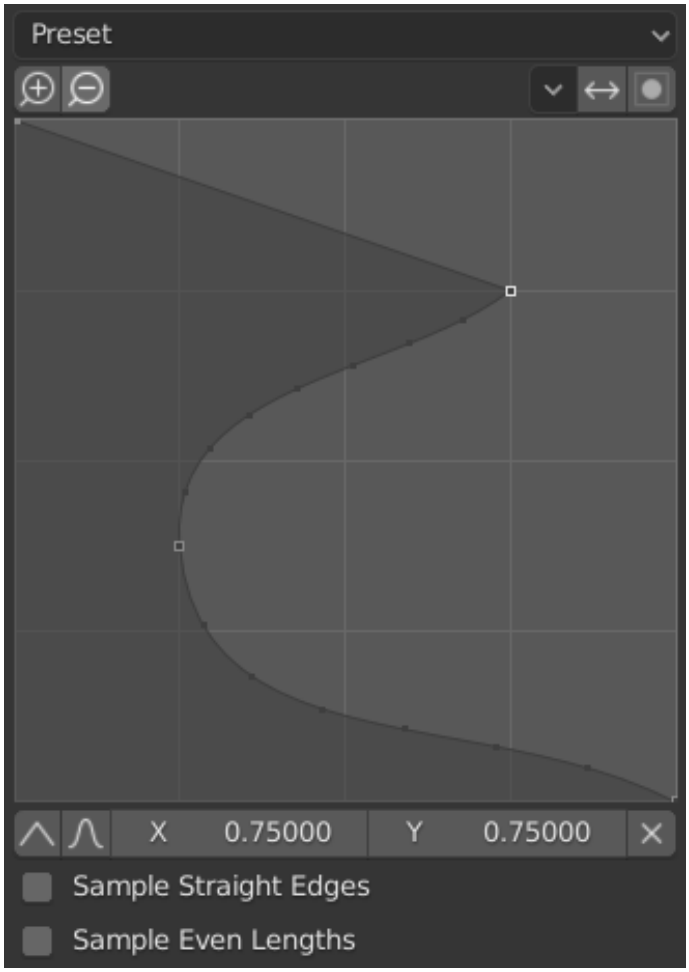

Рис. 22. Вікно налаштувань функції Custom Profile операції Bevel у середовищі Blender® (джерело: https://docs.blender.org/).

На основі аналізу середовища Blender® на предмет створення фасок, було вирішено, що при розробці програмного продукту адаптивного створення фасок буде відтворено метод роботи з розміром фаски – *Percent*.

2.4 Загальні недоліки існуючих рішень для створення фасок

Як вже було зазначено, основним недоліком операції Bevel у більшості середовищ є відсутність адаптивності алгоритму. Проте окрім специфічних проблем середовищ існують також глобальні недоліки методу в цілому.

Так, операція Bevel на полігональному об’єкті з рефлективним матеріалом може бути причиною артефактів пов’язаних з екранним згладжуванням (Anti-Aliasing[6]).

Якщо дивитися на об’єкт з великої відстані, рефлекс може виглядати рвано, зубчасто та не чітко. Ця проблема спільна для усієї растрової графіки, тому на практиці, для отримання рендеру рефлективних поверхонь без артефактів, необхідно задавати значення атрибуту *Width* трохи більше за реальне. Ця проблема також може бути вирішена використанням іншого алгоритму Anti-Aliasing, наприклад Temporal AA[7].

Недоліком також можна вважати збільшення кількості полігонів. Для відеокарти збільшення кількості полігонів означає більше навантаження та більше часу необхідного на рендер. З першого погляду Bevel незначно змінює кількість полігонів, проте у випадку Realtime[8] рендеру, навіть такі зміни можуть призвести до нестабільної роботи. Тому в відеоіграх активно використовують технологію Normal Map[9].

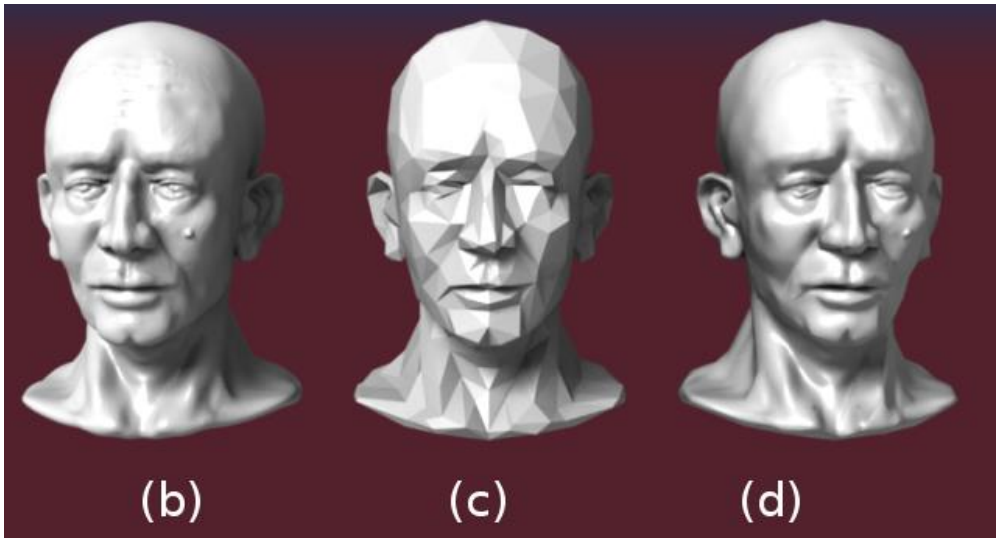

Рис. 23. Приклад роботи технології normal map.

2.5 Можливості операції для адаптивного створення фасок

Вхідними даними алгоритму адаптивного створення фасок являються полігональний об’єкт та набір параметрів:

1. *Scale (тип даних: float)* – радіус проведення операції, визначає загальний розмір фаски.

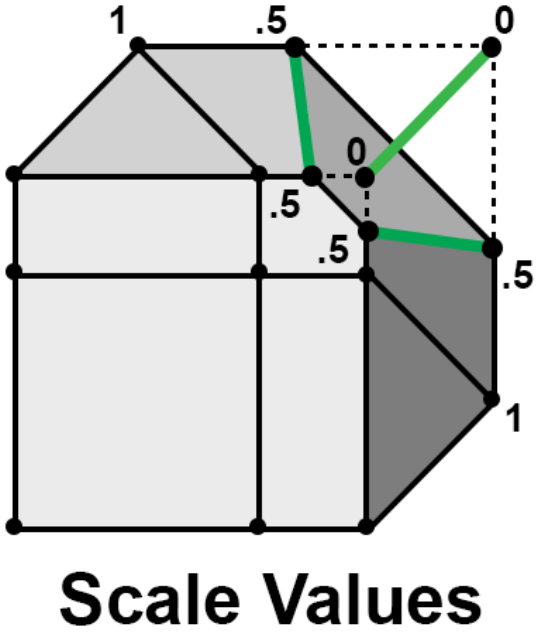

Рис. 24. Схематичне зображення властивості атрибута scale.

2. *Segments (тип даних: int)* – кількість додаткових, паралельних ребер.

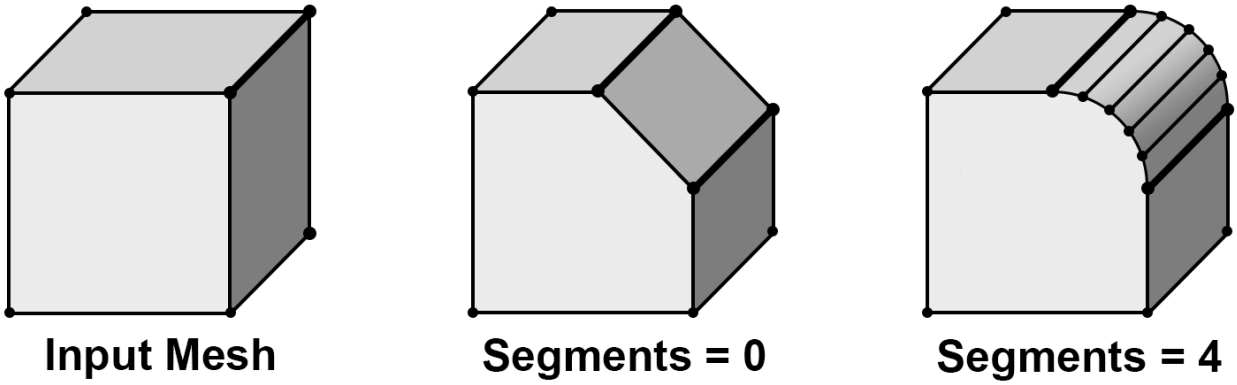

Рис. 25. Схематичне зображення властивості атрибута segments.

3. *Depth (тип даних: curve)* – управляє напрямком кривої вигину при *Segments > 0*, де додатковий об’єкт типу *Curve* контролює вигин.

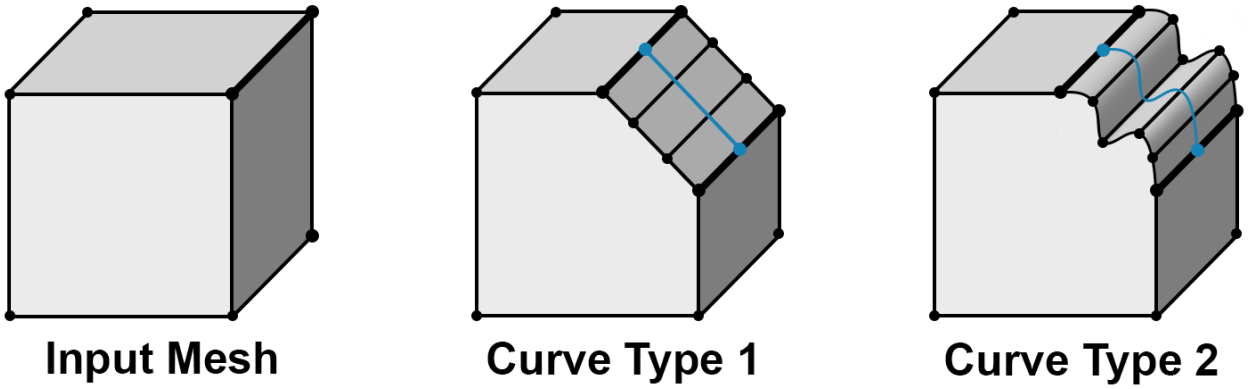

Рис. 26. Схематичне зображення властивості атрибута depth.

4. *Taper (тип даних: float)* – управляє розміром паралельного вигину нових граней.

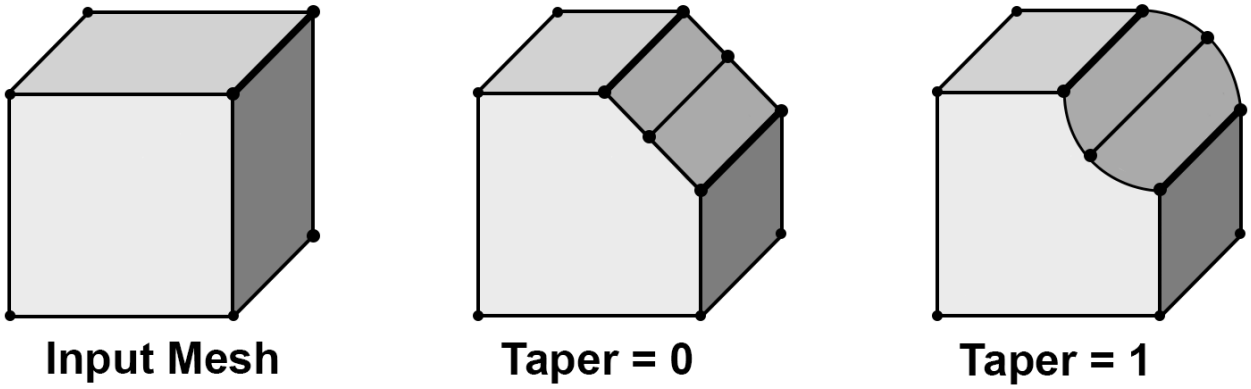

Рис. 27. Схематичне зображення властивості атрибута taper.

Вихідними даними алгоритму являється полігональний об’єкт ідентичний вхідному, за виключенням опрацьованих ребер.

3 ЗАСОБИ РОЗРОБКИ ПРОГРАМНОГО ПРОДУКТУ В СЕРЕДОВИЩІ AUTODESK® MAYA®

Середовище Autodesk® Maya® надає два інтерфейси для розробки плагіну: використовуючи *Maya Commands[10]* або *Maya API[11]*.

Maya Commands:

1. MEL – (Maya Embedded Language) це потужна скриптова мова програмування, яка використовується для виклику внутрішніх команд Maya®.
2. Python[12] — не менш потужна, самостійна мова програмування, яка також може використовуватися для виклику внутрішніх команд Maya®.

Maya API:

1. C++ API (Application Programmer Interface) – забезпечує найкращу продуктивність роботи серед усіх інших варіантів. За допомогою C++ API можна додавати нові об’єкти до середовища Maya®. Також дозволяє викликати MEL команди через API.
2. Python API – заснований на C++ API, дозволяє працювати з API використовуючи мову програмування Python. Існує дві версії Python API: Python API 1.0 та Python API 2.0.
3. .NET API – аналогічно з Python API, дозволяє працювати з API використовуючи Microsoft .NET. Більшість назв та інтерфейсів класів .NET API співпадають з класами C++ API.

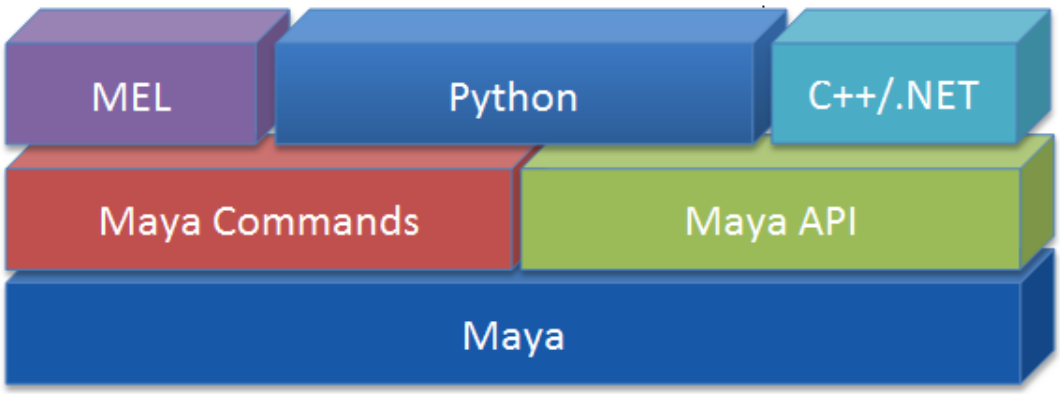

Рис. 28. Ієрархія середовища Autodesk® Maya® (джерело: http://help.autodesk.com/).

На низькому рівні, Maya® — це спеціалізована база даних для зберігання графічної інформації. Така база даних називається Dependency Graph (DG). Інформація в DG зберігається в об'єктах, що називаються нодами. Ноди мають властивості — атрибути, що зберігають характеристики кожної ноди. Атрибути з спільними типами можуть бути з'єднані між собою, таким чином передаючи дані з одної ноди в іншу.

Розглянемо як приклад ноду pCylinderShape та pCubeShape. Ці ноди містять в собі інформацію про полігональні примітиви: циліндр та куб. Обидві ноди мають атрибути «In Mesh» та «Out Mesh». Створимо додаткову ноду polySmooth для демонстрації роботи зав’язків в DG. З’єднаємо атрибут «Out Mesh» ноди pCylinderShape з атрибутом «Input Polymesh» ноди polySmooth, в свою чергу атрибут «Output» ноди polySmooth з’єднаємо з атрибутом «In Mesh» ноди pCubeShape. Таким чином працює нодова система.

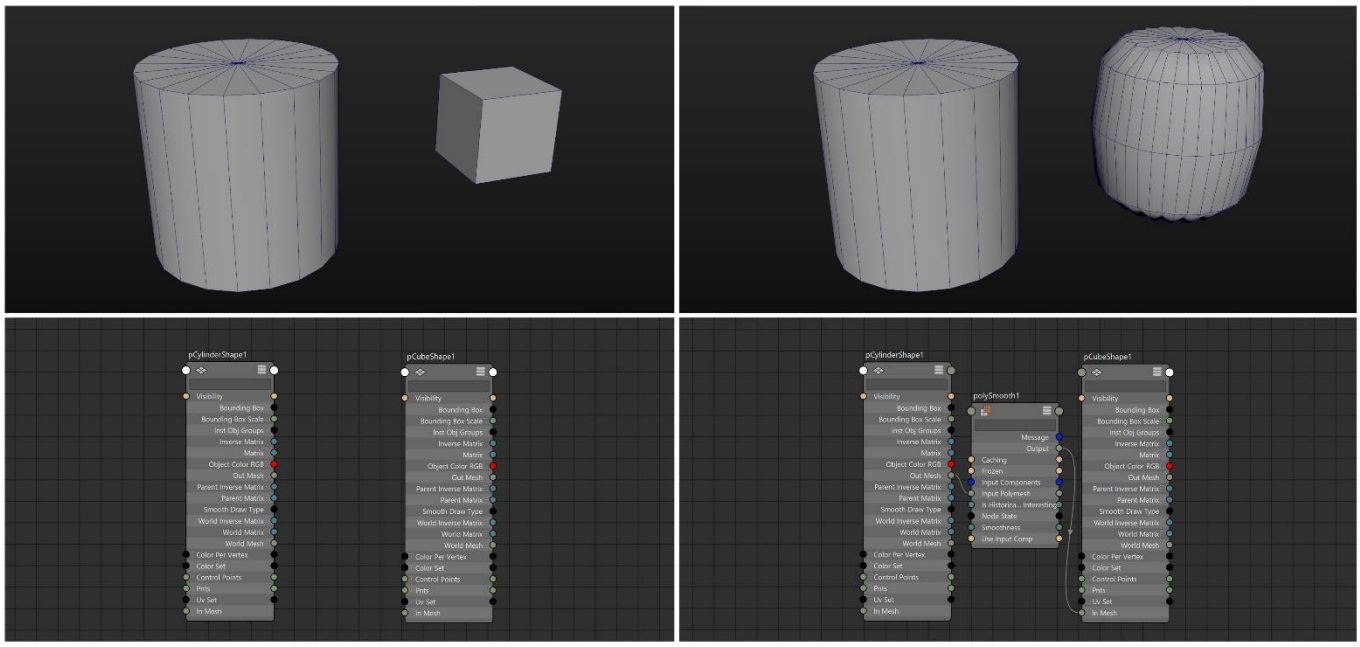

Рис. 29. Приклад роботи залежностей в нодовій системі Autodesk® Maya® .

Ця система в першу чергу дозволяє впровадити систему історії, або *Construction History (CH)*. При зміні одного з вхідних *атрибутів* на будь-якій *ноді*, *DG* перераховує лише ті частини історії, які впливають на розрахунок поверхні.

Робота безпосередньо на рівні *DG* дає максимальну гнучкість, проте сильно ускладнює процес. Тому на високому рівні, для швидких рішень можна працювати на

рівні команд. За замовчуванням Maya® має понад 900 вбудованих команд, що оперують з готовими нодами, встановлюють і з'єднують їх атрибути. Maya API надає доступ до вагомої частини функцій Maya®, недоступних іншим інтерфейсам. В доступний функціонал входять: можливість модифікувати існуючі дані такі, як геометрія, transform ноди, ієрархії, графи сцен, та DG ноди.

У випадку алгоритму дипломної роботи, сценарій потребує використання *Maya Python API* та *Maya Commands*.

4 АРХІТЕКТУРА ПРОГРАМНОГО ПРОДУКТУ

4.1 Загальна архітектура програмного продукту

Архітектура Maya®, за допомогою Python API дозволяє створювати логічні блоки – ноди, та команди. Нода, динамічно отримує на вхід необхідну інформацію про полігональний об’єкт, проводить обчислення використовуючи атрибути, та повертає результуючий об’єкт. Команда, створює ноду, передає флаги в її атрибути та налаштовує зв’язки.

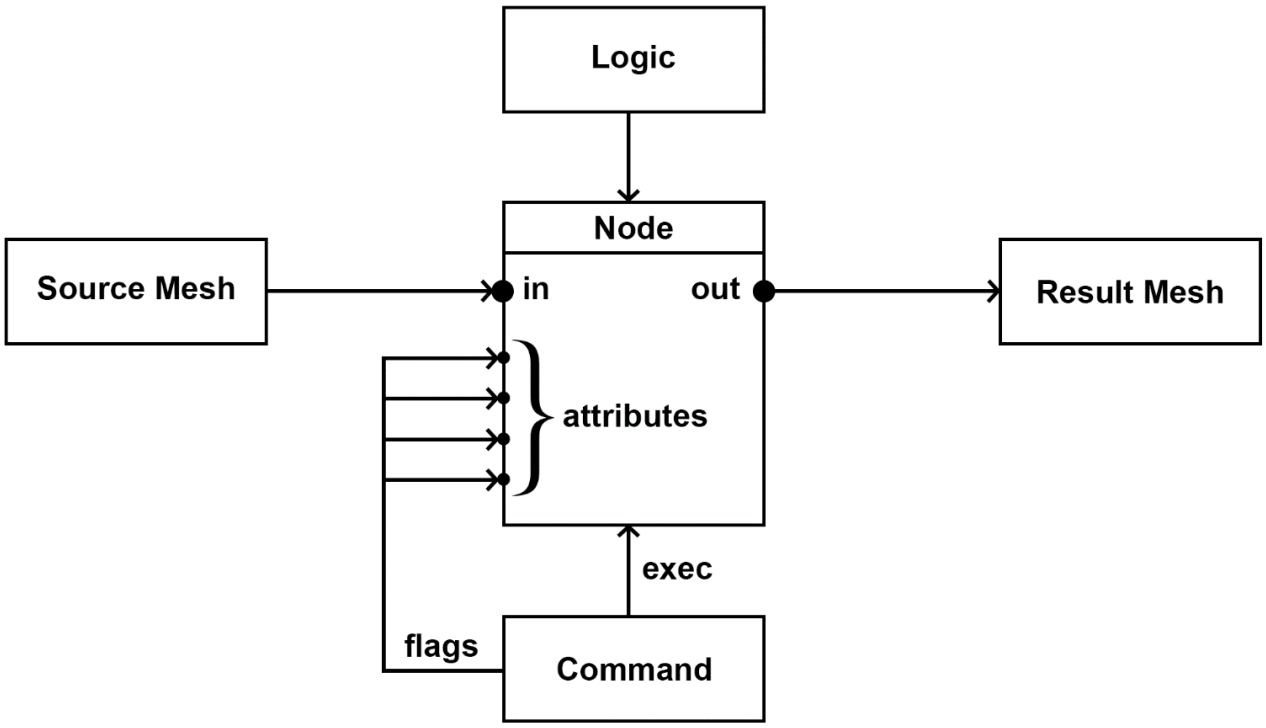

Рис. 30. Схематичне зображення взаємодії логічних елементів плагіну .

Нода використовується для обробки вхідних даних, і по суті виконує логіку операції. У виапдку ноди *adaptiveBevel*, основними, первинними атрибутами (без яких виконання операції неможливе) являються:

1. «In Mesh» - вхідний полігональний об’єкт.
2. «Out Mesh» - вихідний полігональний об’єкт.

Вторинними атрибутами являються внутрішні параметри, що використовуються під час обробки логіки операції. При зміні будь-якого вхідного атрибута, виконується повний перерахунок ланцюжка.

Команду можна викликати за допомогою Script Editor[13]. Команда використовується для швидкого створення ноди зі зв’язками та атрибутами, і має вигляд

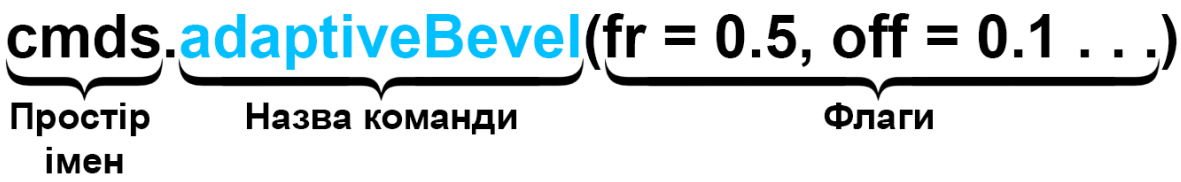

Рис. 31. Синтаксис команди.

При виконанні команди (із Script Editor або за допомогою кнопки на Shelf) створюється нода adaptiveBevel, а значення флагів передаються у відповідні атрибути ноди. Для того, щоб при виконанні команди встановити зв'язок новоствореної ноди з іншими нодами, що приймають участь в СН, використовується бібліотека polyModifier.py авторства Autodesk®.

4.2 Використання сторонніх бібліотек

polyModifierCmd – це загальний, базовий клас розроблений для допомоги в розробці плагінів для модифікації полігональних об’єктів. Полігональні об’єкти в Maya® мають дві особливості:

- Construction History – система історії в Autodesk® Maya®.
- Tweaks – ручні модифікації компонентів на полігональних об’єктах, наприклад переміщення точки.

Обидві особливості мають серйозний вплив на структуру об’єкту та на подальші маніпуляції з ним. Для розуміння який саме вплив має СН та Tweaks на полігональний об’єкт, необхідно ввести поняття стану ноди.

Щоб визначити стан ноди щодо CH та Tweaks, необхідно провести три перевірки:

1. Перевірку на існування Construction History?
2. Перевірку на існування Tweaks?
3. Перевірку на стан Construction History (увімкнена чи вимкнена)?

Результат кожної з цих перевірок змінює спосіб інтерпретації полігонального об’єкту. Що впливає на те, як можна отримати доступ або змінити сітку полігонального об’єкту.

У першому випадку, CH сповіщає нас про те, що існує один лінійний ланцюг вищестоящих нод у DG. Це ланцюг історії. У верхній частині ланцюга є «оригінальний» полігональний об’єкт, а у нижній – «фінальний» полігональний об’єкт, де «оригінальний» та «фінальний» представляють відповідні стани ноди, щодо CH. Кожна з цих нод приєднується через атрибути «inMesh» та «outMesh». Тепер, при зміні ноди інформація буде завжди записуватися в атрибут «inMesh». Проте маємо проблему. Якщо історія існує, то будь-які зміни, внесені до мережі будуть відкинуті. Тож, внесення змін до полігонального об’єкту напряму, неможливо, коли історія існує. Щоб коректно змінити полігональний об’єкт, що має CH, необхідно ввести концепцію ноди модифікатора (*polyModifierNode*). Така нода інкапсулює операції над сіткою, та веде себе аналогічно іншим нодам в CH.

Рис. 32. Стан DG до та після модифікації.

У другому випадку, Tweaks зберігаються у скритому атрибуті полігонального об’єкту. За замовчуванням, DG додає значення tweak до атрибуту «inMesh». З цього випливає, що вставка ноди модифікатора наступним елементом ланцюга після ноди полігонального об’єкту – приводить до зміни порядку проведення операцій, що може призвести до некоректного розрахунку результуючого полігонального об’єкту. Для

обходу даної проблеми, необхідно отримати значення Tweaks, видалити їх з полігонального об’єкту та вставити в ланцюг як ноду *polyTweak* з отриманим значенням. Таким чином конфлікт порядку операцій вирішений:

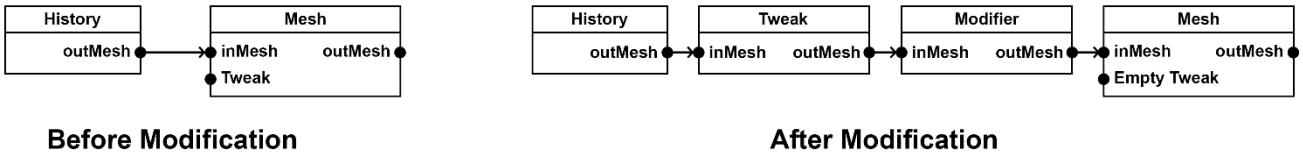

Рис. 33. Стан DG до та після модифікації.

Останньою перевіркою є стан CH. Це задає як буде проводитися модифікація ноди та як нода буде виглядати у ланцюгу DG.

Якщо CH увімкнена, то операція буде проходити по алгоритму описаному вище. Проте, якщо CH вимкнена, то ланцюг історії буде відсутній, то існує два варіанти модифікації полігонального об’єкту:

1. Виконати операцію напряму.
2. Використовувати ланцюг DG як описано вище, але останнім кроком поєднати всі ноди в один полігональний об’єкт.

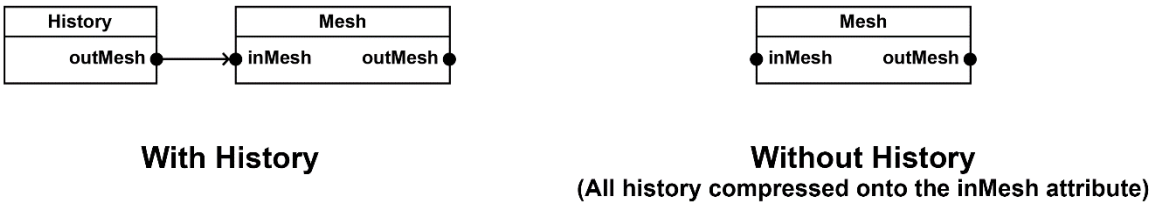

Рис. 34. Стан DG з увімкненою та вимкненою CH .

polyModifierCmd працює з різноманітними станами нод, схожим чином на те, як Maya® опрацьовує CH та Tweaks у полігональних об’єктах. Важливим зауваженням буде те, що CH існує незалежно від Tweaks, та навпаки. Вони ніяк не впливають одне на одного (з точки зору їхнього стану). Розглянемо кожен випадок окремо:

Для History існує 4 випадки, які необхідно опрацьовувати:

Рис. 35. Чотири можливі стани нод.

Для випадків *a)* і *b)*, команда обробляє ноду однаково. Не зважаючи на те, увімкнена СН або вимкнена, якщо СН вже існує на ноді, необхідно опрацьовувати ноду так, ніби СН увімкнена. В такому випадку команда виконує наступний алгоритм:

1) Створити ноду-модифікатор.
2) Знайти найближчу ноду, що напряму зв’язана з нодою полігонального об’єкту.
3) Від’єднати знайдену ноду від ноди полігонального об’єкту.
4) З’єднати знайдену ноду з нодою-модифікатором.
5) З’єднати ноду-модифікатор з нодою полігонального об’єкту.

Для випадку *c)*, *polyModifierCmd* має згенерувати вхідний полігональний об’єкт для керування нодою-модифікатором. Для цього, створюється дублікат ноди полігонального об’єкту, що з’єднується так само як найближча зв’язана нода у попередніх випадках:

1) Створити ноду-модифікатор.
2) Створити дублікат ноди полігонального об’єкту.
3) З’єднати дублікат ноди полігонального об’єкту з нодою-модифікатором.
4) З’єднати ноду-модифікатор з нодою полігонального об’єкту.

Для випадку *d)*, команда складніша. Існує два варіанта обробки у випадку якщо СН не використовується взагалі.

Перший варіант – використати аналогічний до випадку *c)* підхід. Проте це може стати серйозною перешкодою для коректної роботи методів *undo()* та *redo()*, тому, що Maya API не підтримує таких маніпуляцій з СН.

Другий варіант – це просто не вносити ніяких даних до СН та проводити операцію напряму з полігональним мешем.

При обробці Tweaks, аналогічно до History, існує 4 випадки, які необхідно опрацьовувати:

a) History (yes) – RecordHistory (yes)
b) History (yes) – RecordHistory (no)
c) History (no) – RecordHistory (yes)
d) History (no) – RecordHistory (no)

Рис. 36. Чотири можливі стани нод.

Для *a)*, *b)* та *c)* маємо наступний алгоритм:

1) Створити tweak ноду.
2) Дістати tweaks з ноди полігонального об’єкту.
3) Скопіювати tweaks на tweak ноду.
4) Стерти tweaks з ноди полігонального об’єкту.
5) Стерти tweaks з дубліката ноди полігонального об’єкту (лише для *c)* випадку!)

Для випадку *d)*, команда складніша. Існує два варіанта обробки у випадку якщо Tweaks не використовується взагалі.

Перший варіант – використати аналогічний до випадку *c)* підхід. Проте це може стати серйозною перешкодою для коректної роботи методів *undo()* та *redo()*. Другий варіант – це просто не вносити ніяких даних до СН та проводити операцію напряму з полігональним мешем.

5 ОПИС АЛГОРИТМУ СТВОРЕННЯ АДАПТИВНИХ ФАСОК

Створення алгоритму можна поділити на 4 основні кроки:

1. Збір та упорядкування необхідних даних.
2. Створення нових точок та робота з нормалями.
3. Знаходження нових координат точок.
4. Створення заповнюючих граней.

5.1 Збір та упорядкування необхідних даних

Перш за все у класі *adaptiveBevelCmd* необхідно ініціалізувати змінні атрибутів, що будуть приймати значення флагів передані при виклику команди. Якщо флаг не викликається необхідно передбачити значення за замовчуванням.

Далі необхідно створити масив ребер і точок, що підлягають обробці. Для цього здійснюється пошук вибраних компонентів на вхідному полігональному об’єкті. Оскільки алгоритм може опрацьовувати тільки ребра, одразу після створення масиву обраних компонентів виділення має бути конвертовано:

1. Якщо масив компонентів складається з ребер, зміни не вносяться.
2. Якщо масив компонентів складається з точок, маємо додаткову умову:
    - Якщо довжина масиву більша за одиницю, компоненти конвертуються використовуючи команду *ConvertSelectionToContainedEdges*.
    - Якщо довжина масиву менша за одиницю, компоненти конвертуються використовуючи команду *ConvertSelectionToEdges*.
3. Якщо масив компонентів складається з граней, компоненти конвертуються використовуючи команду *ConvertSelectionToEdgePerimeter*.

На цьому збір необхідних даних не завершений. Проте без модифікації полігонального об’єкту, на даному етапі більше корисних даних не знайти. Тому до збору інформації алгоритм буде опосередковано повертатися під час наступних етапів.

5.2 Створення нових точок та робота з нормалями

Після проведення конвертації, масив компонентів складається лише з ребер, що дозволяє розпочати вносити зміни до полігонального об’єкту. В подальшому, всі логічні операції будуть проводитися з точками.

У різних точок на полігональному об’єкті може бути різна кількість нормалей *(vertex normal)*, в залежності від кількості поєднаних в точці граней, що лежать в різних площинах.

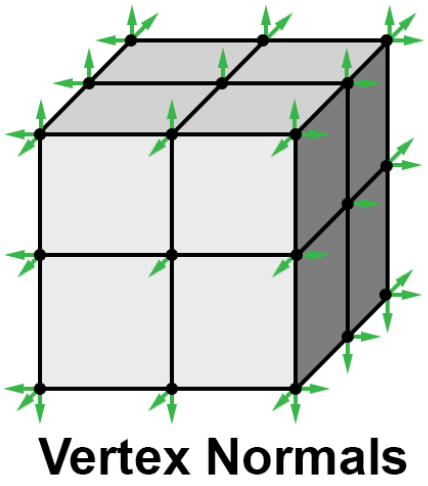

Рис. 37. Схематичне зображення нормалей точок простого полігонального об’єкту.

Щоб опрацювати точку з декількома *(n)* нормалями, необхідно на її місці створити *n* точок, в кожній з яких буде одна унікальна нормаль з оригінальної точки. Для цього необхідно використати операції *«polySplitEdge*[14]*», «polySplitVertex*[15]*», «DetachComponent»*. Проте дані методи коректно працюють тільки на лупах (edge loops[16]).

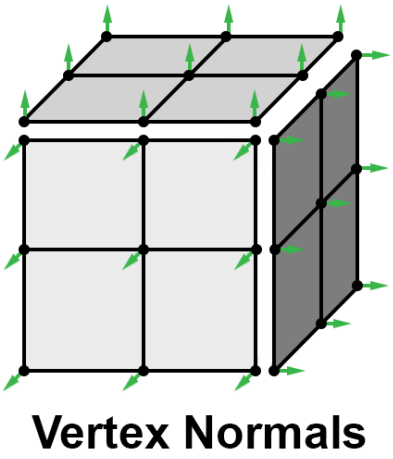

Рис. 38. Схематичне зображення нормалей точок після проведення операцій «polySplitEdge», «polySplitVert», «DetachComponent».

5.3 Знаходження нових координат точок

Наступною частиною алгоритму являється пошук нових координат для кожної точки, що приймає участь в операції. Для цього необхідно опрацьовувати кожну точку окремо, враховуючи її відношення до сусідніх точок. Реалізується дана концепція за допомогою вкладеного циклу в якому здійснюється пошук сусідніх елементів точки для подальшого обрахунку.

Для більшої гнучкості було вирішено розділити стандартний та адаптивний алгоритми операції *adaptiveBevel*. Таким чином наступним кроком після знаходження сусідніх елементів до точки являється перевірка на обраний метод проведення операції. За замовчуванням, тобто якщо флаг вибору методу відсутній операція проводиться *default* методом, проте за допомогою флагу обрати можна також і *adaptive* метод.

- *Default* – аналог стандартної операції *polyBevel3* в середовищі *Autodesk® Maya®*.
- *Adaptive* – адаптивний метод описаний в попередніх розділах.

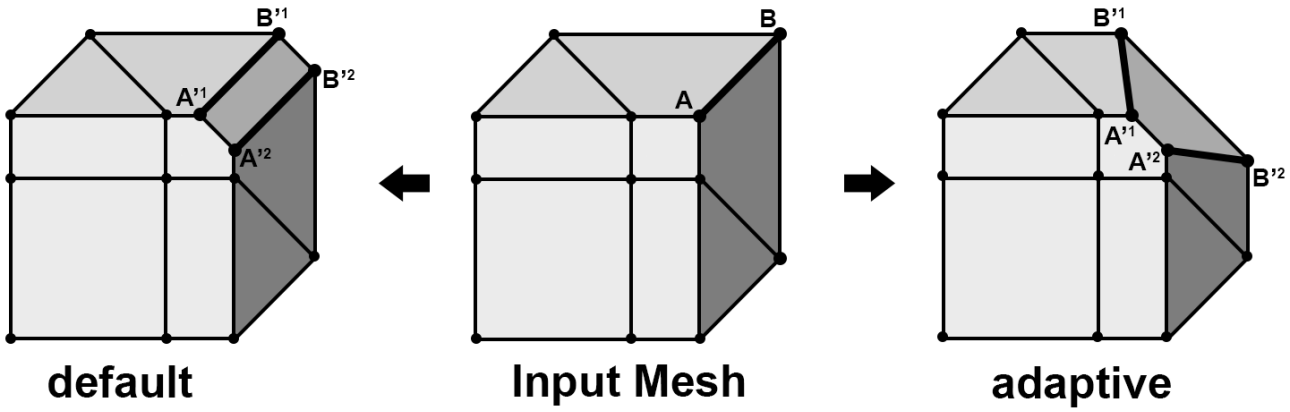

Рис. 39. Схематичне зображення методів роботи операцій adaptiveBevel.

Незалежно від обраного методу, на даному етапі вводиться поняття *pullDirection.*

pullDirection – це вектор, що вказує напрям у якому буде рухатися точка під час проведення операції *adaptiveBevel*.

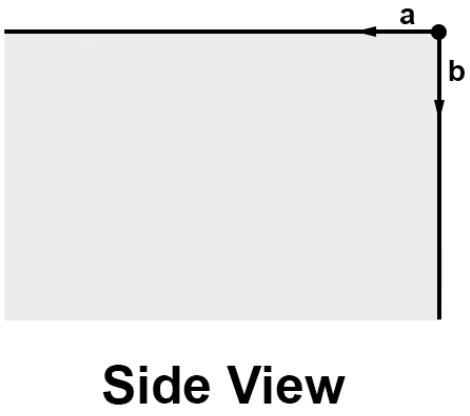

Рис. 40. Схематичне зображення векторів pullDirection.

Вектори *a* та *b* являються векторами *pullDirection* для двох точок, з однаковими координатами. Алгоритм пошуку *pullDirection* буде відрізнятися відповідно до положення сусідніх до шуканих точок. Саме для цього необхідно ввести поняття, та розділити ребра на вирівняні *(aligned)* та не вирівняні *(not aligned)*.

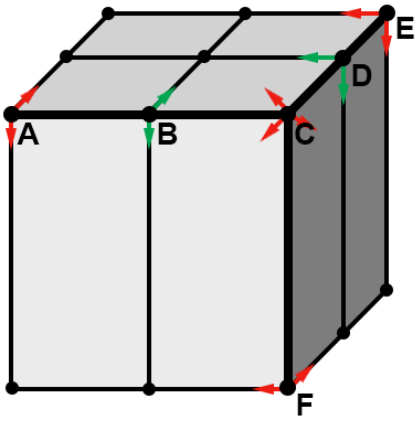

Рис. 41. Схематичне зображення векторів pullDirection.

Ребра вважаються вирівняними, якщо вони лежать на одному прямому відрізку в *3D* просторі. Таким чином ребра *(AB і BC) і (CD і DE)* – вирівняні, а ребра *BC і CD* – ні.

Якщо ребра вирівняні *(aligned)*, то їх векторний добуток[17] дорівнює 0, оскільки векторний добуток двох векторів – це вектор перпендикулярний до цих обох векторів. Для того, щоб знайти векторний добуток спочатку необхідно знайти координати векторів.

$$\text{baVectorPos} = \begin{bmatrix} aVertexPos_x - bVertexPos_x \\ aVertexPos_y - bVertexPos_y \\ aVertexPos_z - bVertexPos_z \end{bmatrix} \quad (1)$$

$$\text{bcVectorPos} = \begin{bmatrix} cVertexPos_x - bVertexPos_x \\ cVertexPos_y - bVertexPos_y \\ cVertexPos_z - bVertexPos_z \end{bmatrix} \quad (2)$$

За формулою векторного добутку:

$$\text{baDpBc} = \begin{bmatrix} baVectorPos_y \cdot bcVectorPos_z - baVectorPos_z \cdot bcVectorPos_y \\ baVectorPos_z \cdot bcVectorPos_x - baVectorPos_x \cdot bcVectorPos_z \\ baVectorPos_x \cdot bcVectorPos_y - baVectorPos_y \cdot bcVectorPos_x \end{bmatrix} \quad (3)$$

Таким чином маємо умову:

$$if\ \text{baDpBc} = [|\text{baDpBc}_x| = 0]\ and\ \left[\left|\text{baDpBc}_y\right| = 0\right]\ and\ [|\text{baDpBc}_z| = 0]$$

Якщо умова виконана, то *pullDirection* дорівнює векторному добутку нормалі в точці перетину ребер і вектору *AC*.

$$pullDirection\ = Bn\ \times AC \quad (4)$$

Проте проблемою можуть стати паралельні до шуканих ребра. Оскільки фактично їх векторний добуток буде ідентичним, необхідна додаткова перевірка для того, щоб інвертувати їх значення.

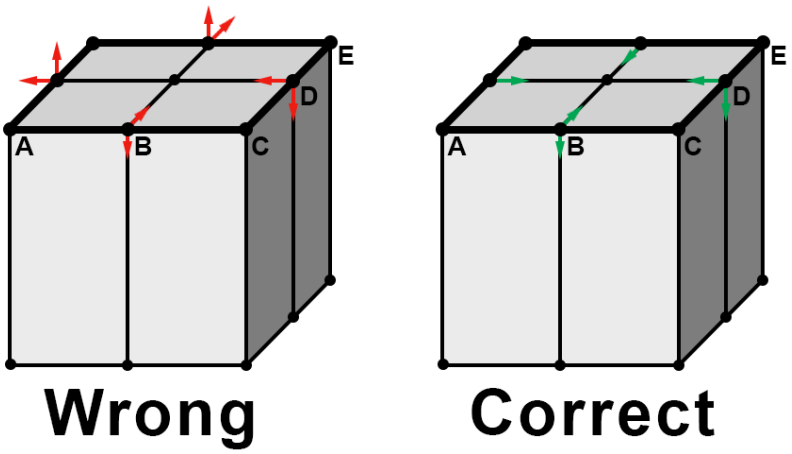

Рис. 42. Схематичне зображення різниці між коректними та некоректними векторами pullDirection.

Для перевірки необхідно знайти три сусідні до шуканої точки *P* та записати в масив.

Назвемо ці три точки *A, B і C* відповідно.

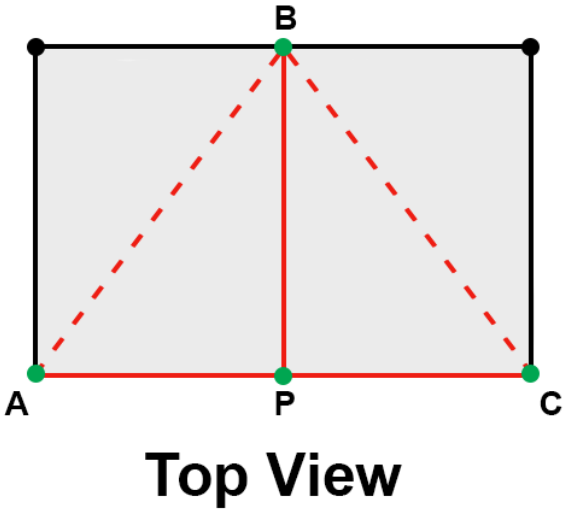

Рис. 43. Схематичне зображення перевірочних трикутників.

Далі необхідно розрахувати координати та величини векторів: *AB, AP, BC, BP, CA та CP*. Координати для *BA* та *BC* рахуються за формулами *(1)* та *(2)*, відповідно, а величини за формулою:

$$\text{abVectorMod} = \sqrt{abVectorPos_x{}^2 + abVectorPos_y{}^2 + abVectorPos_z{}^2} \quad (5)$$

Коли всі координати та величини розраховані, знайдемо площі внутрішніх трикутників *ABP та PBC*. Спочатку розрахувавши периметри:

$$p_1 = \frac{1}{2} \cdot (abVectorMod + bpVectorMod + apVectorMod) \quad (6)$$

$$p_2 = \frac{1}{2} \cdot (bcVectorMod + cpVectorMod + bpVectorMod) \quad (7)$$

За формулою площі:

$$s_1 = \sqrt{p_1 \cdot (p_1 - abVectorMod) \cdot (p_1 - bpVectorMod) \cdot (p_1 - apVectorMod)} \quad (8)$$

$$s_2 = \sqrt{p_2 \cdot (p_2 - bcVectorMod) \cdot (p_2 - cpVectorMod) \cdot (p_2 - bpVectorMod)} \quad (9)$$

Наступним кроком буде розрахунок площі s_3 головного трикутника *ABC*.

$$s_3 = \sqrt{p_3 \cdot (p_3 - abVectorMod) \cdot (p_3 - bcVectorMod) \cdot (p_3 - cфVectorMod)} \quad (10)$$

Таким чином маємо умову:

$$if\ |s_3 - (s1 + s2)| < eps, \text{де } eps \sim 0.0001$$

Якщо умова виконується, то *pullDirection* залишається незмінним. В іншому випадку, якщо умова не виконана, необхідно розрахувати *pullDirection* за оберненою формулою *(4)*.

$$pullDirection = AC \times Bn \quad (11)$$

Якщо ребра не вирівняні *(not aligned)*, то необхідно знайти вектор, напрямлений по бісектрисі кута між $\overrightarrow{BA}$ і $\overrightarrow{BC}$.

Вектор $\overrightarrow{BA} + \overrightarrow{BC}$ – діагональ паралелограма, яка не являється бісектрисою кута. Якщо, ж вектори привести до норм, зробивши їх довжини однаковими, то маємо ромб, у якого діагональ буде бісектрисою. Такі вектори при додаванні дадуть необхідний напрямок. Знайдемо норму векторів:

$$\text{baVectorE} = \left[\frac{baVectorPos_x}{\text{baVectorMag}}, \frac{baVectorPos_y}{\text{baVectorMag}}, \frac{baVectorPos_z}{\text{baVectorMag}}\right] \quad (12)$$

$$\text{bcVectorE} = \left[\frac{bcVectorPos_x}{\text{bcVectorMag}}, \frac{bcVectorPos_y}{\text{bcVectorMag}}, \frac{bcVectorPos_z}{\text{bcVectorMag}}\right] \quad (13)$$

За формулою діагоналі ромба:

$$bisector = \begin{bmatrix} baVectorE_x + bcVectorE_x \\ baVectorE_y + bcVectorE_y \\ baVectorE_z + bcVectorE_z \end{bmatrix} \quad (14)$$

Проте, якщо кут між *BA і BC* більший за 180°, то *pullDirection* буде дорівнювати оберненій бісектрисі. Щоб дізнатися чи потрібно інвертувати *pullDirection,* скористаємося формулою мішаного добутку[18].

$$(BA \times BC) \cdot Bn \quad (15)$$

Якщо $(BA \times BC) \cdot Bn < 0$, то *pullDirection* необхідно інвертувати.

Щоб розрахувати нові координати точки, ви користовуємо формулу:

$$newPos = oldPos + pullDirection \cdot scale, \text{де } scale - \text{значення атрибуту}$$

5.4 Створення заповнюючих граней

Для створення заповнюючих граней, необхідно одразу після проведення операції *«polySplitEdge», «polySplitVertex»* та *«DetachComponent»*, створити масив ребер, що мають однакові координати. Для цього необхідно опрацьовувати кожну точку окремо, враховуючи її відношення до всіх інших точок, що приймають участь в операції. Реалізується дана концепція за допомогою вкладеного циклу.

Спершу, необхідно обчислити координати кожного ребра за формулою:

$$[B_x - A_x; B_y - A_y; B_z - A_z]$$

Далі необхідно знайти парні ребра, що мають однакові координати та занести їх у словник. В майбутньому, коли *newPos* буде знайдено, та всі точки будуть переміщені в нові положення, над усіма парними ребрами буде виконана операція *polyBridgeEdge* з вказаними атрибутами.

6 РОБОТА КОРИСТУВАЧА З ПРОГРАМНИМ ЗАБЕЗПЕЧЕННЯМ

6.1 Завантаження плагіну

Існує два шляхи «завантаження» та «вивантаження» плагінів в середовищі *Autodesk® Maya®*. Найшвидшим та найлегшим шляхом є використання *Plug-in Manager*.

Щоб завантажити плагін використовуючи *Plug-in manager* необхідно:

1) Перейти на вкладку *Window* > *Settings/Preferences* > *Plug-in Manager*.
2) Відкрити вікно *Plug-in Manager*, щоб вивести список всіх доступних плагінів.
3) Знайти необхідний плагін.
4) Натиснути на кнопку *«Loaded»* для того, щоб *Maya®* завантажила його.

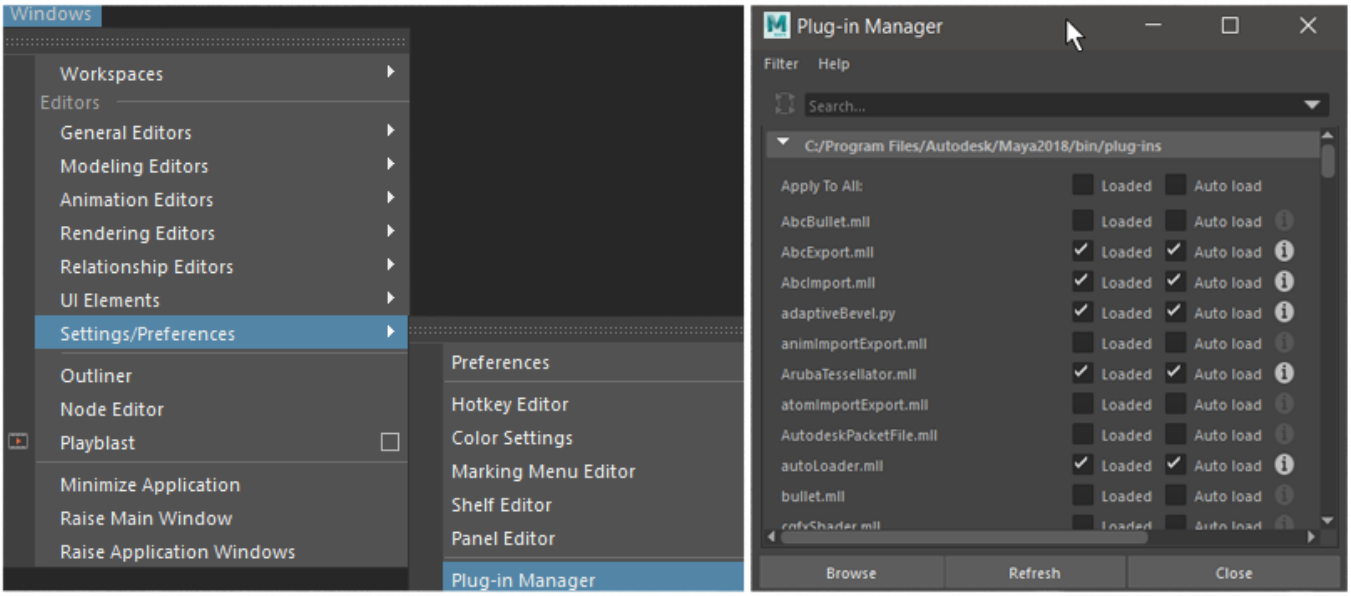

Рис. 44. Навігація до Plug-in Manager.

Plug-in Manager використовує *MAYA_PLUG_IN_PATH* для знаходження наявних для завантаження плагінів.

MAYA_PLUG_IN_PATH сканується при кожному відкритті вікна *Plug-in Manager*. Тому, якщо новий плагін був створений в *MAYA_PLUG_IN_PATH* поки

Maya® працювала, він не буде відображений в *Plug-in Manager*. Для того, щоб нові плагіни відображувались необхідно:

1. Натиснути кнопку *«Refresh»* у вікні *Plug-in Manager* для оновлення списку. При цьому *MAYA_PLUG_IN_PATH* буде повторно просканово на наявність змін.
2. Використати *MEL* команду *loadPlugin.*
3. Перезавантажити *Maya*®.
4. Завантажити плагін з командної строки.

У випадку плагіну *«adaptiveBevel»* в *MAYA_PLUG_IN_PATH, MEL* команду *loadPlugin* необхідно використовувати так:

loadPlugin "adaptiveBevel.py";

Таким чином проводиться пошук файлу з назвою *«adaptiveBevel»* в *MAYA_PLUG_IN_PATH*. Як тільки файл знайдено, він буде завантажений в *Maya*® як плагін.

6.2 Вивантаження плагіну

Вивантажити плагін за допомогою *MEL* можна аналогічно до завантаження, тобто використовуючи команду *unloadPlugin*.

Важливі зауваження:

- Перед тим як вносити зміни до плагіну, він має бути вивантажений. Інакше *Maya*® може видати фатальну помилку.
- Перед тим як вивантажити плагін, спершу необхідно видалити всі елементи що використовують його функціонал з активної сцени. На ряду з видаленням нод плагіну зі сцени, необхідно очистити ноди та команди з *CH* активної сцени, бо вони зберігаються для реалізації методів *undo()* та *redo().*
- При грубому вивантаженню плагіна, що використовується в сцені, перезавантажити ноди плагіну буде неможливо. Причина полягає в тому, що всі існуючі ноди на команди плагіна будуть конвертовані в *«Unknown»* ноди, і при перезавантаженні доступу до зміни їх типу не буде надано.

6.3 Демонстрація роботи програмного забезпечення

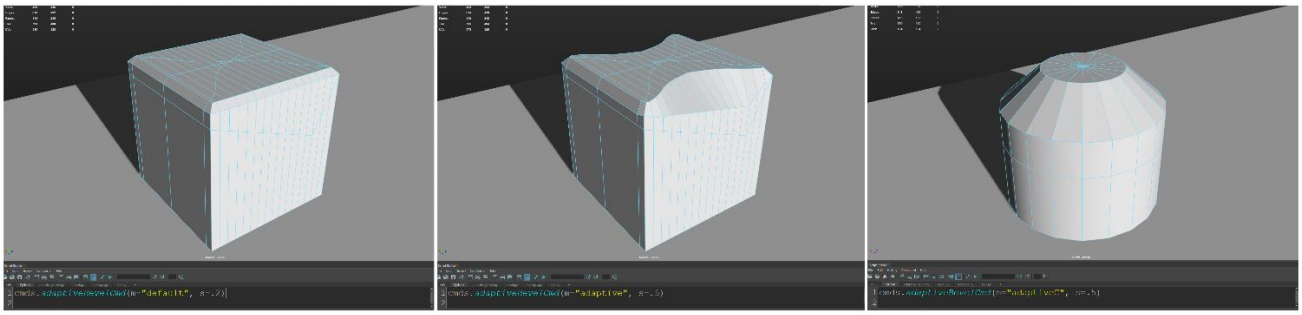

Рис. 45. Демонстрація роботи флагу method.

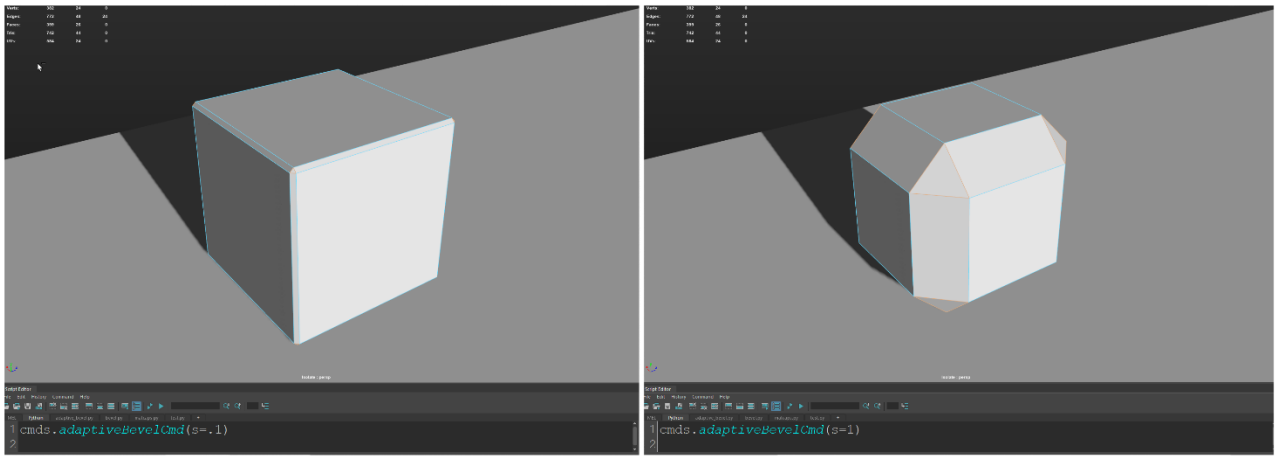

Рис. 46. Демонстрація роботи флагу scale.

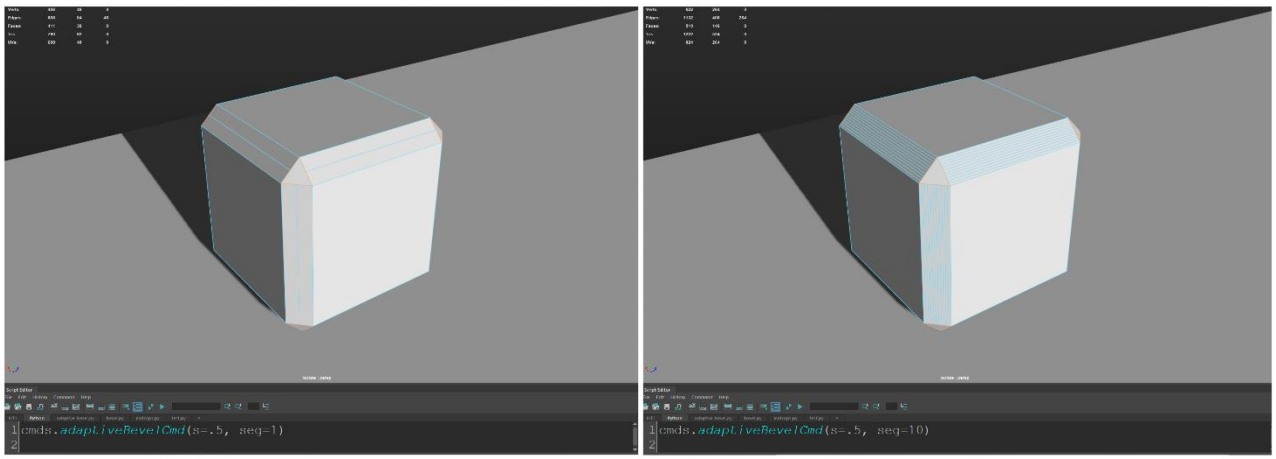

Рис. 47. Демонстрація роботи флагу segments.

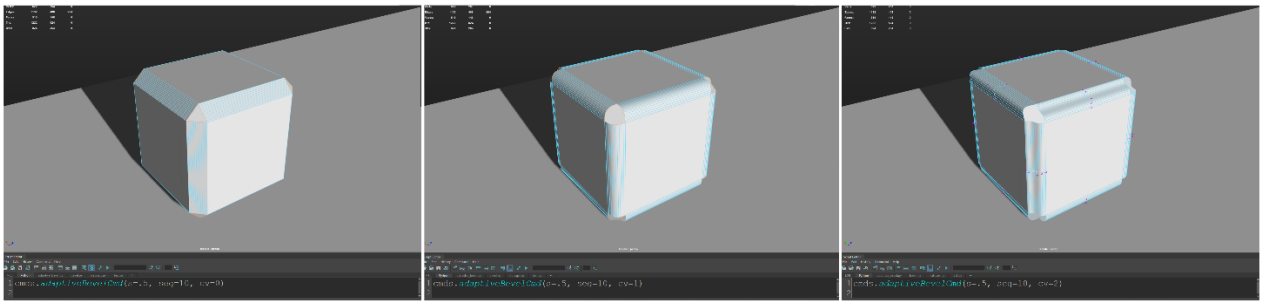

Рис. 48. Демонстрація роботи флагу curve.

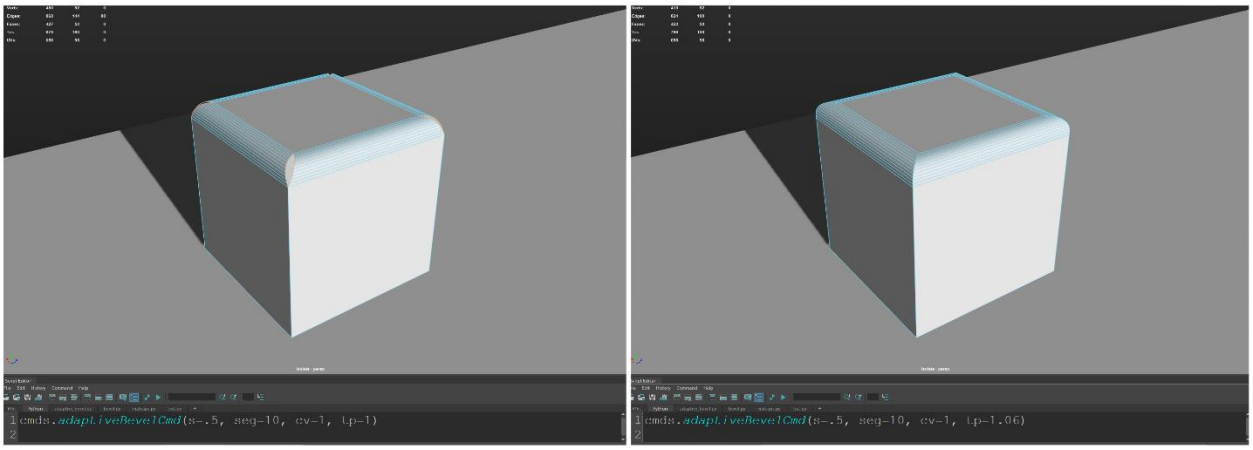

Рис. 49. Демонстрація роботи флагу taper.

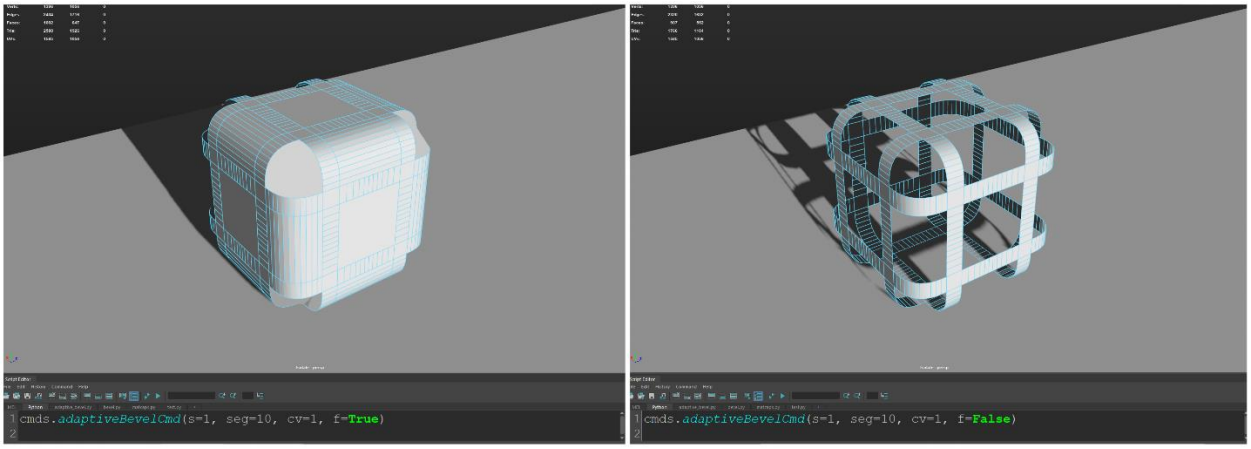

Рис. 50. Демонстрація роботи флагу fill.

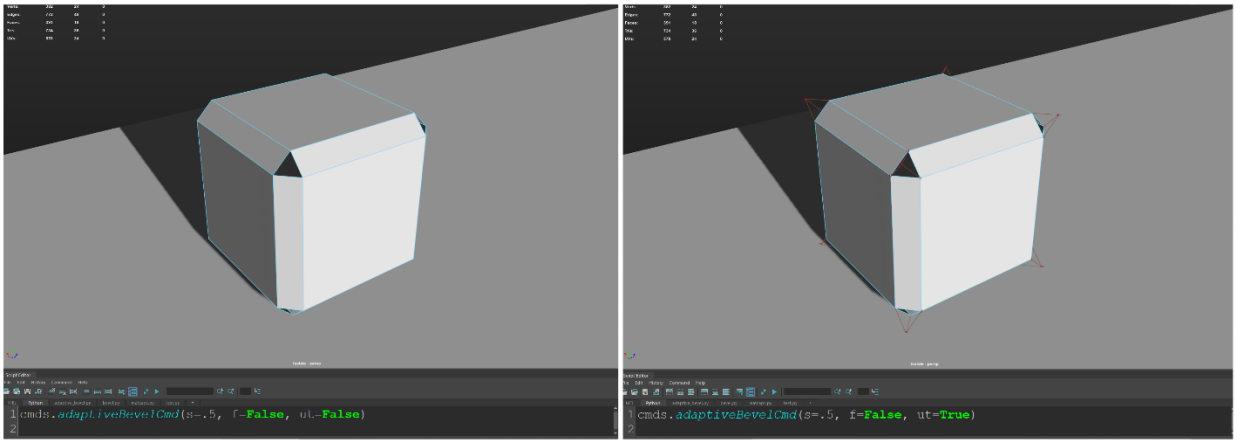

Рис. 51. Демонстрація роботи флагу utility.

ВИСНОВКИ

Дипломну роботу присвячено одній з найпопулярніших операцій в полігональному моделюванні, а саме *Bevel*. При вирішенні поставлених задач отримано наступні результати:

1. Проаналізовані існуючі програмні засоби такі, як: *Autodesk® 3ds Max® і Blender.*
2. В ході роботи було проведено огляд та зроблено аналіз засобів, що були використані для створення даного програмного забезпечення (середовища розробки *PyCharm 2020*, бібліотеки *polyModifiers*, *Maya Python API*, *Maya Commands* та відповідних бібліотек мови *Python*).
3. Створено структуру програмного забезпечення, а саме архітектуру плагіну, що використовує *Maya API* для створення команди та ноди та бібліотеки *polyModifiers* для встановлення зв'язків між ними. Завдяки чому, також, можливе подальше створення інтерфейсу користувача за допомогою *Maya Commands*;
4. Розроблено алгоритмічну базу для реалізації операції *Bevel*;
5. Розроблено програмний продукт, а саме плагін, для створення фасок.

Плагін було написано на мові програмування *Python* з використанням *Maya Python API та Maya Commands*, в середовищі *PyCharm 2020*.

В ході виконання даної роботи було розроблено покращений алгоритм операції *Bevel* для *Autodesk® Maya® 2018*.

Плагін можна використовувати як для створення косметичних фасок так і для імітації інженерних фасок.

Для роботи з даним програмним забезпеченням необхідний комп'ютер середньої потужності та доступ до середовища *Autodesk® Maya® 2018*.

Користувач має змогу створювати фаски та налаштовувати параметри операції *adaptiveBevel*. Також власноруч вносити зміни до висхідного коду плагіну, таким чином створюючи модифікації, щоб задовільнити специфічні потреби. Користувач може створювати адаптивні фаски, уникаючи обмеження стандартної операції *polyBevel3*.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. E. Catmull, J. Clark Recursively generated B-spline surfaces on arbitrary topological meshes / Edwin Catmull, James Clark., 1978. – 350 с.
2. Autodesk. polyBevel3 command. [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: http://help.autodesk.com/cloudhelp/2016/ENU/Maya-Tech-Docs/Nodes/polyBevel3.html
3. Autodesk. Working in Viewport 2.0. [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2018/ENU/Maya-Rendering/files/GUID-677E53F2-E63B-4078-8896-959B3B9BC3AB-htm.html
4. Autodesk. Bevel Modifier. [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2019/ENU/3DSMax-Modifiers/files/GUID-5D4C63C0-E88B-46AB-8EEF-B5BDE836AB48-htm.html
5. Blender Foundation. Bevel Modifier. [Електронний ресурс]. – 2017. – Режим доступу до ресурсу: https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/bevel.html
6. Анализ методов сглаживания на основе super-sampling. [Електронний ресурс]. – 2000. – Режим доступу до ресурсу: https://www.ixbt.com/video/fsaa-an-1.html
7. Introducing Temporal Anti-Aliasing. [Електронний ресурс]. – 2017. – Режим доступу до ресурсу: https://sketchfab.com/blogs/community/introducing-temporal-anti-aliasing/
8. E. Haines, T. Akenine-Moller and N. Hoffman Real-Time Rendering / Eric Haines, Tomas Akenine-Moller, Naty Hoffman., 2008.
9. Blinn. Simulation of Wrinkled Surfaces. – Siggraph, 1978.
10. Autodesk. Maya Commands Reference. [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://help.autodesk.com/cloudhelp/2018/ENU/Maya-Tech-Docs/Commands/index.html
11. Autodesk. Maya Python API 2.0 Reference. [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://help.autodesk.com/view/MAYAUL/2018/ENU/?guid=__py_ref_index_html
12. Adam Mechtley, Ryan Trowbridge Maya Python for Games and Film: A Complete Reference for the Maya Python API: 2011.

13. Autodesk. Script Editor. [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2018/ENU/Maya-Scripting/files/GUID-7C861047-C7E0-4780-ACB5-752CD22AB02E-htm.html
14. Autodesk. polySplitEdge command. [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://help.autodesk.com/cloudhelp/2018/ENU/Maya-Tech-Docs/CommandsPython/polySplitEdge.html
15. Autodesk. polySplitVertex command. [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://help.autodesk.com/cloudhelp/2018/ENU/Maya-Tech-Docs/CommandsPython/polySplitVertex.html
16. Wings 3D. Edge Loops. [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: http://www.wings3d.com/?page_id=766
17. The cross product. [Електронний ресурс]. – Режим доступу до ресурсу: https://mathinsight.org/cross_product
18. The scalar triple product. [Електронний ресурс]. – Режим доступу до ресурсу: https://mathinsight.org/scalar_triple_product
19. Maya Architecture Overview. [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: http://help.autodesk.com/view/MAYAUL/2017/ENU/?guid=__files_GUID_F584322E_4A12_4995_9F4E_A38D4331808F_htm
20. David Gould Complete Maya Programming: An Extensive Guide to MEL and C++ API: 2003.

ДОДАТОК А

Побудова тривимірних моделей з використанням технологій Maya

Специфікація

УКР.НТУУ”КПІ імені Ігоря Сікорського”_ТЕФ_АПЕПС_ТР62

Аркушів 2

Київ 2020

Позначення	Найменування	Примітки
Документація		
УКР.НТУУ”КПІ ”_ТЕФ_АПЕПС_ТР62	Записка.docx	Текстова частина дипломної роботи
Компоненти		
УКР.НТУУ”КПІ ”_ТЕФ_АПЕПС_ТР62 12-1	polyModifier.py	Бібліотека для створення команди та ноди
УКР.НТУУ”КПІ ”_ТЕФ_АПЕПС_ТР62 12-2	adaptiveBevel.py	Виконуваний файл плагіну

ДОДАТОК Б

Побудова тривимірних моделей з використанням технологій Maya

Текст програми

УКР.НТУУ”КПІ імені Ігоря Сікорського”_ТЕФ_АПЕПС_ТР62
12-1

Аркушів 10

Київ 2020

```
import maya.OpenMaya as om
import maya.OpenMayaMPx as ommpx
import maya.cmds as cmds
import math

import polyModifier

class adaptiveBevelCmd(ommpx.MPxCommand):
    kPluginCmdName = 'adaptiveBevelCmd'

    kMethodFlag = '-m'
    kMethodLongFlag = '-method'

    kScaleFlag = '-s'
    kScaleLongFlag = '-scale'

    kSegmentsFlag = '-seg'
    kSegmentsLongFlag = '-segments'

    kCurveFlag = '-cv'
    kCurveLongFlag = '-curve'

    kTaperFlag = '-tp'
    kTaperLongFlag = '-taper'

    kSmoothAngleFlag = '-sa'
    kSmoothAngleLongFlag = '-smooth'

    kFillHoleFlag = '-f'
    kFillHoleLongFlag = '-fill'

    kUtilityFlag = '-ut'
    kUtilityLongFlag = '-utility'

    def __init__(self):
        ommpx.MPxCommand.__init__(self)

    def doIt(self, args):
        try:
            argData = om.MArgDatabase(self.syntax(), args)  # if this fails, it will raise its own exception...
        except:
            pass
        else:
            # Initializing arguments
            if argData.isFlagSet(adaptiveBevelCmd.kMethodFlag):
                BEVEL_METHOD = argData.flagArgumentString(adaptiveBevelCmd.kMethodFlag, 0)
            else:
                # default, adaptive, adaptiveC
                BEVEL_METHOD = "default"
            if argData.isFlagSet(adaptiveBevelCmd.kScaleFlag):
                SCALE = argData.flagArgumentDouble(adaptiveBevelCmd.kScaleFlag, 0)
            else:
                SCALE = .5
            if argData.isFlagSet(adaptiveBevelCmd.kSegmentsFlag):
                SEGMENTS = argData.flagArgumentInt(adaptiveBevelCmd.kSegmentsFlag, 0)
            else:
                SEGMENTS = 0
            if argData.isFlagSet(adaptiveBevelCmd.kCurveFlag):
                CURVE_TYPE = argData.flagArgumentInt(adaptiveBevelCmd.kCurveFlag, 0)
            else:
                CURVE_TYPE = 0
            if argData.isFlagSet(adaptiveBevelCmd.kTaperFlag):
                TAPER = argData.flagArgumentDouble(adaptiveBevelCmd.kTaperFlag, 0)
            else:
                # 1.06
                TAPER = 1
            if argData.isFlagSet(adaptiveBevelCmd.kFillHoleFlag):
                FILL_HOLE = argData.flagArgumentBool(adaptiveBevelCmd.kFillHoleFlag, 0)
            else:
                FILL_HOLE = True
            if argData.isFlagSet(adaptiveBevelCmd.kUtilityFlag):
                DRAW_UTILITY = argData.flagArgumentBool(adaptiveBevelCmd.kUtilityFlag, 0)
            else:
                DRAW_UTILITY = False

            # GLOBAL
```

```
PI = 3.14159

# Epsilons
BRIDGE_EPSILON = .001
BEVEL_EPSILON_ANG = .001
if (BEVEL_METHOD == "default") or (BEVEL_METHOD == "adaptive"):
    BEVEL_ACC_EPSILON = .0001
elif (BEVEL_METHOD == "adaptiveC"):
    BEVEL_ACC_EPSILON = .01

# Conversion of selection to edges (algorithm only works with edges)
def selectionCorrection(selection):
    if cmds.selectType(q=True, eg=True):
        pass
    elif cmds.selectType(q=True, v=True):
        if len(selection) > 1:
            cmds.warning("Multiple vertices selected: selection will be converted to contained edges")
            cmds.ConvertSelectionToContainedEdges(selection)
            if cmds.selectType(q=True, v=True):
                cmds.ConvertSelectionToEdges(selection)
        else:
            cmds.warning("Only one vertex selected")
            cmds.ConvertSelectionToEdges(selection)
    elif cmds.selectType(q=True, fc=True):
        cmds.warning("Face/s selected: selection will be converted to edge perimeter")
        cmds.ConvertSelectionToEdgePerimeter(selection)

# Collapse similar and converse "2 key dictionary" elements
def optimizeDict(dict, key1, key2):
    flag = True
    unique_dict = []
    unique_dict.append(dict[0])
    for i in range(1, len(dict)):
        for j in range(0, i):
            if (dict[j] != dict[i]) and (dict[j][key1] != dict[i][key2]) and (
                    dict[j][key2] != dict[i][key1]):
                continue
            flag = False
            break
        if (flag):
            unique_dict.append(dict[i])
    return unique_dict

# Calculating shared point
def isPointSame(centerVertex, index, jndex):
    if index == jndex:
        centerVertex.append(index)
        return True
    else:
        return False

# Modifying list to get rid of repetitive elements
def unique(list):
    unique_list = []
    for x in list:
        if x not in unique_list:
            unique_list.append(x)
    return unique_list

# Converting radians to degrees
def radToDeg(rad):
    deg = (rad * 180) / PI
    return deg

# "Love triangle"
def loveTriangler(triangle, bVertexPos, pullingDirection):
    aLTVertex = triangle[0]
    bLTVertex = triangle[1]
    cLTVertex = triangle[2]

    pVertexPos = [bVertexPos[0] + pullingDirection[0] * .05, bVertexPos[1] + pullingDirection[1] * .05,
                  bVertexPos[2] + pullingDirection[2] * .05]

    # Calculating AB segment of triangle
    abVectorPos = [cmds.pointPosition(bLTVertex)[0] - cmds.pointPosition(aLTVertex)[0],
                   cmds.pointPosition(bLTVertex)[1] - cmds.pointPosition(aLTVertex)[1],
```

```
                    cmds.pointPosition(bLTVertex)[2] - cmds.pointPosition(aLTVertex)[2]]
        abVectorMod = math.sqrt((abVectorPos[0] ** 2) + (abVectorPos[1] ** 2) + (abVectorPos[2] ** 2))

        # Calculating AP segment of triangle
        apVectorPos = [pVertexPos[0] - cmds.pointPosition(aLTVertex)[0],
                       pVertexPos[1] - cmds.pointPosition(aLTVertex)[1],
                       pVertexPos[2] - cmds.pointPosition(aLTVertex)[2]]
        apVectorMod = math.sqrt((apVectorPos[0] ** 2) + (apVectorPos[1] ** 2) + (apVectorPos[2] ** 2))

        # Calculating BC segment of triangle
        bcVectorPos = [cmds.pointPosition(cLTVertex)[0] - cmds.pointPosition(bLTVertex)[0],
                       cmds.pointPosition(cLTVertex)[1] - cmds.pointPosition(bLTVertex)[1],
                       cmds.pointPosition(cLTVertex)[2] - cmds.pointPosition(bLTVertex)[2]]
        bcVectorMod = math.sqrt((bcVectorPos[0] ** 2) + (bcVectorPos[1] ** 2) + (bcVectorPos[2] ** 2))

        # Calculating BP segment of triangle
        bpVectorPos = [pVertexPos[0] - cmds.pointPosition(bLTVertex)[0],
                       pVertexPos[1] - cmds.pointPosition(bLTVertex)[1],
                       pVertexPos[2] - cmds.pointPosition(bLTVertex)[2]]
        bpVectorMod = math.sqrt((bpVectorPos[0] ** 2) + (bpVectorPos[1] ** 2) + (bpVectorPos[2] ** 2))

        # Calculating CA segment of triangle
        caVectorPos = [cmds.pointPosition(aLTVertex)[0] - cmds.pointPosition(cLTVertex)[0],
                       cmds.pointPosition(aLTVertex)[1] - cmds.pointPosition(cLTVertex)[1],
                       cmds.pointPosition(aLTVertex)[2] - cmds.pointPosition(cLTVertex)[2]]
        caVectorMod = math.sqrt((caVectorPos[0] ** 2) + (caVectorPos[1] ** 2) + (caVectorPos[2] ** 2))

        # Calculating CP segment of triangle
        cpVectorPos = [pVertexPos[0] - cmds.pointPosition(cLTVertex)[0],
                       pVertexPos[1] - cmds.pointPosition(cLTVertex)[1],
                       pVertexPos[2] - cmds.pointPosition(cLTVertex)[2]]
        cpVectorMod = math.sqrt((cpVectorPos[0] ** 2) + (cpVectorPos[1] ** 2) + (cpVectorPos[2] ** 2))

        # Calculating inner triangles perimeter
        p1 = .5 * (abVectorMod + bpVectorMod + apVectorMod)
        p2 = .5 * (caVectorMod + apVectorMod + cpVectorMod)
        p3 = .5 * (bcVectorMod + cpVectorMod + bpVectorMod)

        # Calculating main triangle perimeter
        p4 = .5 * (abVectorMod + bcVectorMod + caVectorMod)

        # Calculating inner triangles area
        s1 = math.sqrt(p1 * (p1 - abVectorMod) * (p1 - bpVectorMod) * (p1 - apVectorMod))
        s2 = math.sqrt(p2 * (p2 - caVectorMod) * (p2 - apVectorMod) * (p2 - cpVectorMod))
        s3 = math.sqrt(p3 * (p3 - bcVectorMod) * (p3 - cpVectorMod) * (p3 - bpVectorMod))

        # Calculating main triangle area
        s4 = math.sqrt(p4 * (p4 - abVectorMod) * (p4 - bcVectorMod) * (p4 - caVectorMod))

        if abs(s4 - (s1 + s2 + s3)) < BEVEL_ACC_EPSILON:
            return True
        else:
            return False

            # Draw curve directions of operation

    def drawUtility(vertexPos, newPos):
        startPos = vertexPos
        endPos = newPos
        cmds.curve(p=[startPos, endPos])

    selection = cmds.ls(sl=True, fl=True)
    selectionCorrection(selection)

    selectedComponents = cmds.polySplitEdge(op=1)
    selectedComponents = cmds.ls(sl=True, fl=True)

    # Calculate dictionary of bridge pairs
    bridgeInfos = []
    for i in range(len(selectedComponents)):
        indexVertices = cmds.polyListComponentConversion(selectedComponents[i], tv=True)
        indexVertices = cmds.filterExpand(indexVertices, sm=31)
        for j in range(len(selectedComponents)):
            if j == i:
                pass
            else:
```

```
                        jndexVertices = cmds.polyListComponentConversion(selectedComponents[j], tv=True)
                        jndexVertices = cmds.filterExpand(jndexVertices, sm=31)

                        ac = [abs(cmds.pointPosition(indexVertices[0])[0] -
cmds.pointPosition(jndexVertices[0])[0]),
                              abs(cmds.pointPosition(indexVertices[0])[1] -
cmds.pointPosition(jndexVertices[0])[1]),
                              abs(cmds.pointPosition(indexVertices[0])[2] -
cmds.pointPosition(jndexVertices[0])[2])]
                        bd = [abs(cmds.pointPosition(indexVertices[1])[0] -
cmds.pointPosition(jndexVertices[1])[0]),
                              abs(cmds.pointPosition(indexVertices[1])[1] -
cmds.pointPosition(jndexVertices[1])[1]),
                              abs(cmds.pointPosition(indexVertices[1])[2] -
cmds.pointPosition(jndexVertices[1])[2])]
                        ad = [abs(cmds.pointPosition(indexVertices[0])[0] -
cmds.pointPosition(jndexVertices[1])[0]),
                              abs(cmds.pointPosition(indexVertices[0])[1] -
cmds.pointPosition(jndexVertices[1])[1]),
                              abs(cmds.pointPosition(indexVertices[0])[2] -
cmds.pointPosition(jndexVertices[1])[2])]
                        bc = [abs(cmds.pointPosition(indexVertices[1])[0] -
cmds.pointPosition(jndexVertices[0])[0]),
                              abs(cmds.pointPosition(indexVertices[1])[1] -
cmds.pointPosition(jndexVertices[0])[1]),
                              abs(cmds.pointPosition(indexVertices[1])[2] -
cmds.pointPosition(jndexVertices[0])[2])]

                        if (ac[0] < BRIDGE_EPSILON) and (ac[1] < BRIDGE_EPSILON) and (ac[2] < BRIDGE_EPSILON) and (
                                bd[0] < BRIDGE_EPSILON) and (bd[1] < BRIDGE_EPSILON) and (bd[2] < BRIDGE_EPSILON) or
(
                                ad[0] < BRIDGE_EPSILON) and (ad[1] < BRIDGE_EPSILON) and (ad[2] < BRIDGE_EPSILON)
and (
                                bc[0] < BRIDGE_EPSILON) and (bc[1] < BRIDGE_EPSILON) and (bc[2] < BRIDGE_EPSILON):
                            bridgeInfo = {
                                "first": selectedComponents[i],
                                "second": selectedComponents[j],
                            }
                            bridgeInfos.append(bridgeInfo)

            bridgeInfos = optimizeDict(bridgeInfos, "first", "second")

            # Bevel
            verticesInfo = []
            for i in range(len(selectedComponents)):
                indexVertices = cmds.polyListComponentConversion(selectedComponents[i], tv=True)
                indexVertices = cmds.filterExpand(indexVertices, sm=31)
                for j in range(i + 1, len(selectedComponents)):
                    if i == len(selectedComponents):
                        j = 0
                    jndexVertices = cmds.polyListComponentConversion(selectedComponents[j], tv=True)
                    jndexVertices = cmds.filterExpand(jndexVertices, sm=31)

                    bVertex = []
                    if isPointSame(bVertex, indexVertices[0], jndexVertices[0]) or isPointSame(bVertex,
                                                                                               indexVertices[0],
                                                                                               jndexVertices[
                                                                                                   1]) or
isPointSame(
                            bVertex, indexVertices[1], jndexVertices[0]) or isPointSame(bVertex, indexVertices[1],
                                                                                        jndexVertices[1]):
                        bVertexPos = cmds.pointPosition(bVertex)
                        if indexVertices[0] == bVertex[0]:
                            aVertex = indexVertices[1]
                            aVertexPos = cmds.pointPosition(indexVertices[1])
                        elif indexVertices[1] == bVertex[0]:
                            aVertex = indexVertices[0]
                            aVertexPos = cmds.pointPosition(indexVertices[0])
                        if jndexVertices[0] == bVertex[0]:
                            cVertex = jndexVertices[1]
                            cVertexPos = cmds.pointPosition(jndexVertices[1])
                        elif jndexVertices[1] == bVertex[0]:
                            cVertex = jndexVertices[0]
                            cVertexPos = cmds.pointPosition(jndexVertices[0])

                            # Defining connected vectors (BA & BC)
```

```
                    baVectorPos = [aVertexPos[0] - bVertexPos[0], aVertexPos[1] - bVertexPos[1],
                                   aVertexPos[2] - bVertexPos[2]]
                    bcVectorPos = [cVertexPos[0] - bVertexPos[0], cVertexPos[1] - bVertexPos[1],
                                   cVertexPos[2] - bVertexPos[2]]
                    acVectorPos = [cVertexPos[0] - aVertexPos[0], cVertexPos[1] - aVertexPos[1],
                                   cVertexPos[2] - aVertexPos[2]]

                    # Vectors magnitude
                    baVectorMag = math.sqrt(baVectorPos[0] ** 2 + baVectorPos[1] ** 2 + baVectorPos[2] ** 2)
                    bcVectorMag = math.sqrt(bcVectorPos[0] ** 2 + bcVectorPos[1] ** 2 + bcVectorPos[2] ** 2)

                    # Dot product of 2 connected vectors
                    baDpBc = baVectorPos[0] * bcVectorPos[0] + baVectorPos[1] * bcVectorPos[1] + baVectorPos[2]
* \
                             bcVectorPos[2]

                    # Cross product of 2 connected vectors (BA x BC = perpendicular vector)
                    baCpBc = [baVectorPos[1] * bcVectorPos[2] - baVectorPos[2] * bcVectorPos[1],
                              baVectorPos[2] * bcVectorPos[0] - baVectorPos[0] * bcVectorPos[2],
                              baVectorPos[0] * bcVectorPos[1] - baVectorPos[1] * bcVectorPos[0]]

                    # list of normal vectors of vertex B
                    bn = cmds.polyNormalPerVertex(bVertex, query=True, xyz=True)
                    bn = [bn[i * 3:(i + 1) * 3] for i in range((len(bn) + 3 - 1) // 3)]
                    bn = unique(bn)

                    # If default method selected
                    if BEVEL_METHOD == "default":
                        # isAligned = True
                        if (abs(baCpBc[0]) < BEVEL_EPSILON_ANG) and (abs(baCpBc[1]) < BEVEL_EPSILON_ANG) and (
                                abs(baCpBc[2]) < BEVEL_EPSILON_ANG):
                            bn = bn[0]

                            pullingDirection = [bn[1] * acVectorPos[2] - bn[2] * acVectorPos[1],
                                                bn[2] * acVectorPos[0] - bn[0] * acVectorPos[2],
                                                bn[0] * acVectorPos[1] - bn[1] * acVectorPos[0]]

                            cmds.select(bVertex)
                            cmds.GrowLoopPolygonSelectionRegion()
                            triangle = cmds.ls(sl=True, fl=True)
                            triangle.remove(bVertex[0])

                            if loveTriangler(triangle, bVertexPos, pullingDirection):
                                # cross product of the normal vector of vertex B and vector AC
                                pullingDirection = [bn[1] * acVectorPos[2] - bn[2] * acVectorPos[1],
                                                    bn[2] * acVectorPos[0] - bn[0] * acVectorPos[2],
                                                    bn[0] * acVectorPos[1] - bn[1] * acVectorPos[0]]
                            else:
                                # cross product of the vector AC and normal vector of vertex B
                                pullingDirection = [acVectorPos[1] * bn[2] - acVectorPos[2] * bn[1],
                                                    acVectorPos[2] * bn[0] - acVectorPos[0] * bn[2],
                                                    acVectorPos[0] * bn[1] - acVectorPos[1] * bn[0]]

                            # Making unit vector of pullingDirection
                            pullingDirectionMod = math.sqrt(
                                pullingDirection[0] ** 2 + pullingDirection[1] ** 2 + pullingDirection[2] ** 2)
                            pullingDirection = [pullingDirection[0] / pullingDirectionMod,
                                                pullingDirection[1] / pullingDirectionMod,
                                                pullingDirection[2] / pullingDirectionMod]

                            # Calculating new point position for vertex B
                            newPosition = [bVertexPos[0] + pullingDirection[0] * SCALE * .2,
                                           bVertexPos[1] + pullingDirection[1] * SCALE * .2,
                                           bVertexPos[2] + pullingDirection[2] * SCALE * .2]

                            if DRAW_UTILITY:
                                drawUtility(bVertexPos, newPosition)

                                # isAligned = False
                        else:
                            if len(bn) > 1:
                                pullingDirection = [bn[0][1] * bn[1][2] - bn[0][2] * bn[1][1],
                                                    bn[0][2] * bn[1][0] - bn[0][0] * bn[1][2],
                                                    bn[0][0] * bn[1][1] - bn[0][1] * bn[1][0]]
                            else:
                                bn = bn[0]
```

```
                                # scalar tripple product ((BA x BC) * Bn)
                                sTP = [baCpBc[0] * bn[0] + baCpBc[1] * bn[1] + baCpBc[2] * bn[2]]

                                angle = radToDeg(math.acos(baDpBc / baVectorMag * bcVectorMag))

                                if (abs(sTP[0]) < BEVEL_EPSILON_ANG):
                                    pullingDirection = [bn[1] * acVectorPos[2] - bn[2] * acVectorPos[1],
                                                        bn[2] * acVectorPos[0] - bn[0] * acVectorPos[2],
                                                        bn[0] * acVectorPos[1] - bn[1] * acVectorPos[0]]

                                    cmds.select(bVertex)
                                    cmds.GrowLoopPolygonSelectionRegion()
                                    triangle = cmds.ls(sl=True, fl=True)
                                    triangle.remove(bVertex[0])

                                    if loveTriangler(triangle, bVertexPos, pullingDirection):
                                        # cross product of the normal vector of vertex B and vector BC
                                        pullingDirection = [bn[1] * acVectorPos[2] - bn[2] * acVectorPos[1],
                                                            bn[2] * acVectorPos[0] - bn[0] * acVectorPos[2],
                                                            bn[0] * acVectorPos[1] - bn[1] * acVectorPos[0]]
                                    else:
                                        # cross product of the vector BC and normal vector of vertex B
                                        pullingDirection = [acVectorPos[1] * bn[2] - acVectorPos[2] * bn[1],
                                                            acVectorPos[2] * bn[0] - acVectorPos[0] * bn[2],
                                                            acVectorPos[0] * bn[1] - acVectorPos[1] * bn[0]]
                                else:
                                    bisector = [baVectorMag * baVectorPos[0] + bcVectorPos[0],
                                                baVectorPos[1] + bcVectorPos[1], baVectorPos[2] +
bcVectorPos[2]]


                                    bisectorDpBn = bisector[0] * bn[0] + bisector[1] * bn[1] + bisector[2] *
bn[2]
                                    bisectorMag = math.sqrt(bisector[0] ** 2 + bisector[1] ** 2 + bisector[2] **
2)
                                    bnMag = math.sqrt(bn[0] ** 2 + bn[1] ** 2 + bn[2] ** 2)
                                    angle2 = radToDeg(math.acos(bisectorDpBn / bisectorMag * bnMag))

                                    baVectorE = [baVectorPos[0] / baVectorMag, baVectorPos[1] / baVectorMag,
                                                 baVectorPos[2] / baVectorMag]
                                    bcVectorE = [bcVectorPos[0] / bcVectorMag, bcVectorPos[1] / bcVectorMag,
                                                 bcVectorPos[2] / bcVectorMag]

                                    baCpBcDpBn = baCpBc[0] * bn[0] + baCpBc[1] * bn[1] + baCpBc[2] * bn[2]

                                    bisector = [baVectorE[0] + bcVectorE[0], baVectorE[1] + bcVectorE[1],
                                                baVectorE[2] + bcVectorE[2]]
                                    pullingDirection = bisector

                                    # "Love triangle" checks if pulling direction is correct
                                    cmds.select(bVertex)
                                    cmds.GrowLoopPolygonSelectionRegion()
                                    triangle = cmds.ls(sl=True, fl=True)
                                    triangle.remove(bVertex[0])

                                    if len(triangle) > 2:
                                        if loveTriangler(triangle, bVertexPos, pullingDirection):
                                            # cross product of the normal vector of vertex B and vector BC
                                            pullingDirection = [-bisector[0], -bisector[1], -bisector[2]]
                                        else:
                                            # cross product of the vector BC and normal vector of vertex B
                                            pullingDirection = bisector

                            # Calculating new point position for vertex B
                            newPosition = [bVertexPos[0] + pullingDirection[0] * SCALE * .2,
                                           bVertexPos[1] + pullingDirection[1] * SCALE * .2,
                                           bVertexPos[2] + pullingDirection[2] * SCALE * .2]

                            if DRAW_UTILITY:
                                drawUtility(bVertexPos, newPosition)

                        elif BEVEL_METHOD == "adaptive" or BEVEL_METHOD == "adaptiveC":
                            # isAligned = True
                            if (abs(baCpBc[0]) < BEVEL_EPSILON_ANG) and (abs(baCpBc[1]) < BEVEL_EPSILON_ANG) and (
                                    abs(baCpBc[2]) < BEVEL_EPSILON_ANG) or BEVEL_METHOD == "adaptiveC":
                                cmds.select(bVertex)
```

```
            cmds.ConvertSelectionToEdges()
            adaption = cmds.ls(sl=True, fl=True)

            adaptionNew = []
            for k in range(len(adaption)):
                adaptionNew.append(adaption[k])

            for r in range(len(adaption)):
                for g in range(len(selectedComponents)):
                    if adaption[r] == selectedComponents[g]:
                        adaptionNew.remove(adaption[r])

            if len(adaptionNew) == 1:
                vertices = cmds.polyListComponentConversion(adaptionNew[0], tv=True)
                vertices = cmds.filterExpand(vertices, sm=31)
                pullingDirection = [
                    cmds.pointPosition(vertices[1])[0] - cmds.pointPosition(vertices[0])[0],
                    cmds.pointPosition(vertices[1])[1] - cmds.pointPosition(vertices[0])[1],
                    cmds.pointPosition(vertices[1])[2] - cmds.pointPosition(vertices[0])[2]]

                # "Love triangle" checks if pulling direction is correct
                cmds.select(bVertex)
                cmds.GrowLoopPolygonSelectionRegion()
                triangle = cmds.ls(sl=True, fl=True)
                triangle.remove(bVertex[0])

                if loveTriangler(triangle, bVertexPos, pullingDirection):
                    # cross product of the normal vector of vertex B and vector BC
                    pullingDirection = [
                        cmds.pointPosition(vertices[1])[0] - cmds.pointPosition(vertices[0])[0],
                        cmds.pointPosition(vertices[1])[1] - cmds.pointPosition(vertices[0])[1],
                        cmds.pointPosition(vertices[1])[2] - cmds.pointPosition(vertices[0])[2]]
                else:
                    # cross product of the vector BC and normal vector of vertex B
                    pullingDirection = [
                        cmds.pointPosition(vertices[0])[0] - cmds.pointPosition(vertices[1])[0],
                        cmds.pointPosition(vertices[0])[1] - cmds.pointPosition(vertices[1])[1],
                        cmds.pointPosition(vertices[0])[2] - cmds.pointPosition(vertices[1])[2]]

            # Calculating new point position for vertex B
            newPosition = [bVertexPos[0] + pullingDirection[0] * SCALE,
                           bVertexPos[1] + pullingDirection[1] * SCALE,
                           bVertexPos[2] + pullingDirection[2] * SCALE]

            if DRAW_UTILITY:
                drawUtility(bVertexPos, newPosition)

        # isAligned = False
        else:
            if len(bn) > 1:
                pullingDirection = [bn[0][1] * bn[1][2] - bn[0][2] * bn[1][1],
                                    bn[0][2] * bn[1][0] - bn[0][0] * bn[1][2],
                                    bn[0][0] * bn[1][1] - bn[0][1] * bn[1][0]]
            else:
                bn = bn[0]

                # scalar tripple product ((BA x BC) * Bn)
                sTP = [baCpBc[0] * bn[0] + baCpBc[1] * bn[1] + baCpBc[2] * bn[2]]

                angle = radToDeg(math.acos(baDpBc / baVectorMag * bcVectorMag))

                if (abs(sTP[0]) < BEVEL_EPSILON_ANG):
                    pullingDirection = [bn[1] * acVectorPos[2] - bn[2] * acVectorPos[1],
                                        bn[2] * acVectorPos[0] - bn[0] * acVectorPos[2],
                                        bn[0] * acVectorPos[1] - bn[1] * acVectorPos[0]]

                    # "Love triangle" checks if pulling direction is correct
                    cmds.select(bVertex)
                    cmds.GrowLoopPolygonSelectionRegion()
                    triangle = cmds.ls(sl=True, fl=True)
                    triangle.remove(bVertex[0])

                    if loveTriangler(triangle, bVertexPos, pullingDirection):
                        # cross product of the normal vector of vertex B and vector BC
                        pullingDirection = [bn[1] * acVectorPos[2] - bn[2] * acVectorPos[1],
                                            bn[2] * acVectorPos[0] - bn[0] * acVectorPos[2],
```

```
                                                bn[0] * acVectorPos[1] - bn[1] * acVectorPos[0]]
                                    else:
                                        # cross product of the vector BC and normal vector of vertex B
                                        pullingDirection = [acVectorPos[1] * bn[2] - acVectorPos[2] * bn[1],
                                                            acVectorPos[2] * bn[0] - acVectorPos[0] * bn[2],
                                                            acVectorPos[0] * bn[1] - acVectorPos[1] * bn[0]]

                                else:
                                    bisector = [baVectorMag * baVectorPos[0] + bcVectorPos[0],
                                                baVectorPos[1] + bcVectorPos[1], baVectorPos[2] +
bcVectorPos[2]]

                                    bisectorDpBn = bisector[0] * bn[0] + bisector[1] * bn[1] + bisector[2] *
bn[2]
                                    bisectorMag = math.sqrt(bisector[0] ** 2 + bisector[1] ** 2 + bisector[2] **
2)
                                    bnMag = math.sqrt(bn[0] ** 2 + bn[1] ** 2 + bn[2] ** 2)
                                    angle2 = radToDeg(math.acos(bisectorDpBn / bisectorMag * bnMag))

                                    baVectorE = [baVectorPos[0] / baVectorMag, baVectorPos[1] / baVectorMag,
                                                 baVectorPos[2] / baVectorMag]
                                    bcVectorE = [bcVectorPos[0] / bcVectorMag, bcVectorPos[1] / bcVectorMag,
                                                 bcVectorPos[2] / bcVectorMag]

                                    baCpBcDpBn = baCpBc[0] * bn[0] + baCpBc[1] * bn[1] + baCpBc[2] * bn[2]

                                    bisector = [baVectorE[0] + bcVectorE[0], baVectorE[1] + bcVectorE[1],
                                                baVectorE[2] + bcVectorE[2]]
                                    pullingDirection = bisector

                                    # "Love triangle" checks if pulling direction is correct
                                    cmds.select(bVertex)
                                    cmds.GrowLoopPolygonSelectionRegion()
                                    triangle = cmds.ls(sl=True, fl=True)
                                    triangle.remove(bVertex[0])

                                    if len(triangle) > 2:
                                        if loveTriangler(triangle, bVertexPos, pullingDirection):
                                            # cross product of the normal vector of vertex B and vector BC
                                            pullingDirection = [-bisector[0], -bisector[1], -bisector[2]]
                                        else:
                                            # cross product of the vector BC and normal vector of vertex B
                                            pullingDirection = bisector

                                            # Calculating new point position for vertex B
                                newPosition = [bVertexPos[0] + pullingDirection[0] * SCALE * .1,
                                               bVertexPos[1] + pullingDirection[1] * SCALE * .1,
                                               bVertexPos[2] + pullingDirection[2] * SCALE * .1]

                                if DRAW_UTILITY:
                                    drawUtility(bVertexPos, newPosition)

                            vertexInfo = {
                                "name": bVertex,
                                "newPos": newPosition,
                            }

                            verticesInfo.append(vertexInfo)

                            # Set new point position
                    for i in range(len(verticesInfo)):
                        cmds.move(verticesInfo[i]["newPos"][0], verticesInfo[i]["newPos"][1], verticesInfo[i]["newPos"][2],
                                  verticesInfo[i]["name"], absolute=True)

                    # Create connection faces
                    for i in range(len(bridgeInfos)):
                        cmds.polyBridgeEdge(bridgeInfos[i]["first"], bridgeInfos[i]["second"], ctp=CURVE_TYPE, dv=SEGMENTS,
                                            tp=TAPER)

                    cmds.SelectAll()
                    cmds.polyMergeVertex(d=0.0075)
                    # Fill holes
                    if FILL_HOLE:
                        cmds.SelectAll()
                        cmds.FillHole()
```

```
    @classmethod
    def cmdCreator(cls):
        return ommpx.asMPxPtr(cls())

    @classmethod
    def syntaxCreator(cls):
        syntax = om.MSyntax()
        syntax.addFlag(cls.kMethodFlag, cls.kMethodLongFlag, om.MSyntax.kString)
        syntax.addFlag(cls.kScaleFlag, cls.kScaleLongFlag, om.MSyntax.kDouble)
        syntax.addFlag(cls.kSegmentsFlag, cls.kSegmentsLongFlag, om.MSyntax.kLong)
        syntax.addFlag(cls.kCurveFlag, cls.kCurveLongFlag, om.MSyntax.kLong)
        syntax.addFlag(cls.kTaperFlag, cls.kTaperLongFlag, om.MSyntax.kDouble)
        syntax.addFlag(cls.kSmoothAngleFlag, cls.kSmoothAngleLongFlag, om.MSyntax.kLong)
        syntax.addFlag(cls.kFillHoleFlag, cls.kFillHoleLongFlag, om.MSyntax.kBoolean)
        syntax.addFlag(cls.kUtilityFlag, cls.kUtilityLongFlag, om.MSyntax.kBoolean)
        syntax.useSelectionAsDefault(True)
        syntax.setObjectType(om.MSyntax.kSelectionList, 1, 1)
        return syntax

def initializePlugin(obj):
    plugin = ommpx.MFnPlugin(obj, 'Lobus Yurii', '1.0', 'Any')
    try:
        plugin.registerCommand(adaptiveBevelCmd.kPluginCmdName, adaptiveBevelCmd.cmdCreator,
adaptiveBevelCmd.syntaxCreator)
    except:
        raise Exception('Failed to register command: ' + adaptiveBevelCmd.kPluginCmdName)

def uninitializePlugin(obj):
    plugin = ommpx.MFnPlugin(obj)
    try:
        plugin.deregisterCommand(adaptiveBevelCmd.kPluginCmdName)
    except:
        raise Exception('Failed to unregister command: ' + adaptiveBevelCmd.kPluginCmdName)
```

ДОДАТОК В

Побудова тривимірних моделей з використанням технологій Мауа

Опис програмного коду

УКР.НТУУ”КПІ імені Ігоря Сікорського”_ТЕФ_АПЕПС_ТР62
13-1

Аркушів 2

АНОТАЦІЯ

Розроблена система містить дві компоненти. Перша з них – це бібліотека класів, *polyModifiers.py*, авторства *Autodesk*, що бере на себе весь функціонал роботи зі станами команд та нод, без якої роботи плагіну неможлива.

Другим компонентом є файл *adaptiveBevel.py*, що представляє реалізацію алгоритму адаптивного створення фасок.

Плагін було написано на мові програмування *Python* з використанням *Maya Python API та Maya Commands*, в середовищі *PyCharm 2020*.